WD-xsl-19980818

Extensible Stylesheet Language (XSL)

Version 1.0

World Wide Web Consortium Working Draft 18-August-1998

This version

http://www.w3.org/TR/1998/WD-xsl-19980818
http://www.w3.org/TR/1998/WD-xsl-19980818.xml
http://www.w3.org/TR/1998/WD-xsl-19980818.html
http://www.w3.org/TR/1998/WD-xsl-19980818.pdf

Latest version

http://www.w3.org/TR/WD-xsl

Editors

James Clark *(jjc@jclark.com)* [Tree Construction]
Stephen Deach, Adobe *(sdeach@adobe.com)* [Formatting Objects]

Status of this document

This is a W3C Working Draft for review by W3C members and other interested parties. It is the first working draft of XSL. It is based on the XSL Submission, but differs substantially in many important respects. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. The XSL Working Group will not allow early implementation to constrain its ability to make changes to this specification prior to final release. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress". A list of current W3C working drafts can be found at http://www.w3.org/TR.

Comments may be sent to xsl-editors@w3.org. Public discussion of XSL takes place on the XSL-List mailing list.

Abstract

XSL is a language for expressing stylesheets. It consists of two parts:

1. a language for transforming XML documents, and

2. an XML vocabulary for specifying formatting semantics.

An XSL stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary.

Table of Contents

Appendices

1. Overview

XSL is a language for expressing stylesheets. Each stylesheet describes rules for presenting a class of XML source documents. There are two parts to the presentation process. First, the result tree is constructed from the source tree. Second, the result tree is interpreted to produce formatted output on a display, on paper, in speech or onto other media.

The first part, constructing the result tree, 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, the source tree can be filtered and reordered, and arbitrary structure can be added.

The second part, formatting, is achieved by using the formatting vocabulary specified in this document to construct the result tree. Formally, this vocabulary is an XML namespace. Each element type in the vocabulary corresponds to a formatting object class. A formatting object class represents a particular kind of formatting behavior. For example, the block formatting object class represents the breaking of the content of a paragraph into lines. Each attribute in the vocabulary corresponds to a formatting property. A formatting object class has a specific set of formatting properties which provide finer control over the behavior of the formatting object class; for example, controlling indenting of lines, spacing between lines, and spacing before and after the collection of lines. A formatting object can have content, and its formatting behavior is applied to its content.

XSL does not require result trees to use the formatting vocabulary and thus can be used for general XML transformations. For example, XSL can be used to transform XML to "well-formed" HTML, that is, XML that uses the element types and attributes defined by HTML.

When the result tree uses the formatting vocabulary, a conforming XSL implementation must be able to interpret the result tree according to the semantics of the formatting vocabulary as defined in this document; it may also be able to externalize the result tree as XML, but it is not required to be able to do so.

2. Tree Construction

2.1 Overview

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 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 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 **Section 2.5.1: Conflict Resolution for Template Rules**.

XSL uses XML namespaces [W3C XML Names] to distinguish elements that are instructions to the XSL processor from elements that specify literal result tree structure. Instruction elements all belong to the XSL namespace. The examples in this document use a prefix of `xsl:` for elements in the XSL namespace.

An XSL stylesheet contains an `xsl:stylesheet` document element. This element may contain `xsl:template` elements specifying template rules, which will be described later in this document.

The following is an example of a simple XSL stylesheet that constructs a result tree for a sequence of `para` elements containing `emphasis` elements. The `result-ns="fo"` attribute indicates that a tree using the formatting object vocabulary is being constructed. The rule for the root node specifies the use of a page sequence formatted with any font with serifs. The `para` elements become `block` formatting objects which are set in 10 point type with a 12 point space before each block.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/TR/WD-xsl"
  xmlns:fo="http://www.w3.org/TR/WD-xsl/FO"
  result-ns="fo">
  <xsl:template match="/">
    <fo:page-sequence font-family="serif">
       <xsl:process-children/>
    </fo:page-sequence>
  </xsl:template>

  <xsl:template match="para">
    <fo:block font-size="10pt" space-before="12pt">
      <xsl:process-children/>
    </fo:block>
  </xsl:template>
</xsl:stylesheet>
```

The `xsl:stylesheet` element can also contain elements importing other XSL stylesheets, elements defining macros, elements defining global constants, and elements identifying source attributes as individual element identifiers.

2.2 Stylesheet Structure

A stylesheet is represented by an `xsl:stylesheet` element in an XML document.

XSL processors must use the XML namespaces mechanism [W3C XML Names] for both source documents and stylesheets. All XSL defined elements, that is those specified in this document with a prefix of `xsl:`, will only be recognized by the XSL processor if they belong to a namespace with the URI `http://www.w3.org/TR/WD-xsl`; XSL defined elements are recognized only in the stylesheet not in the source document.

> **Issue (versioning):** Should there be some way for a stylesheet to indicate which version of XSL it conforms to? Can this be done through the URI of the XSL namespace?

The `xsl:stylesheet` element has an optional `result-ns` attribute; the value must be a namespace prefix. If this attribute is specified, all result elements must belong to the namespace identified by this prefix. If there is a namespace declared as the default namespace, then an empty string may be used as the value to specify that all result elements belong to that namespace. If the `result-ns` attribute specifies the XSL Formatting Objects namespace, then in addition to constructing the result XML tree, the XSL processor must interpret it according to the semantics defined in this document. The XSL Formatting Objects namespace has the URI `http://www.w3.org/TR/WD-xsl/FO`. The examples in this document use the `fo:` prefix for this namespace.

The `xsl:stylesheet` element may contain the following types of elements:

1. `xsl:import`
2. `xsl:include`
3. `xsl:id`
4. `xsl:strip-space`
5. `xsl:preserve-space`
6. `xsl:define-macro`
7. `xsl:define-attribute-set`
8. `xsl:define-constant`
9. `xsl:template`

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.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:import href="..."/>

  <xsl:include href="..."/>
```

```
  <xsl:id attribute="..."/>

  <xsl:strip-space element="..."/>

  <xsl:preserve-space element="..."/>

  <xsl:define-macro name="...">
   ...
  </xsl:define-macro>

  <xsl:define-attribute-set name="...">
  ...
  </xsl:define-attribute-set>

  <xsl:define-constant name="..." value="..."/>

  <xsl:template match="...">
    ...
  </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.

> **Issue (media-rule):** Should we provide the functionality of CSS's `@media` rule and if so how?

2.3 Processing Model

> **Ed. Note:** This needs expanding and polishing.

A node is processed to create a result tree fragment. The result tree is constructed by processing the root node. 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 for the node. During the instantiation of a template, the node for which the template is being instantiated is called the current node. A template typically contains instructions that select an additional sequence of source nodes for processing. A sequence of source nodes is processed by appending the result tree structure created by processing each of the members of the sequence in order. 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.

2.4 Data Model

XSL operates on an XML document, whether a stylesheet or a source document, as a tree. Any two stylesheets or source documents that have the same tree will be processed the same by XSL. This section describes how XSL models an XML document as a tree. This model is conceptual only and does not mandate any particular implementation.

XML documents operated on by XSL must conform to the XML namespaces specification [W3C XML

Names].

The tree contains nodes and character data. There are three kinds of node:

- root nodes
- element nodes
- attribute nodes

  Issue (node-types): We need to support at least processing instructions and comments, possibly other node types as well. Support requires having patterns that match/select them, being able to get at their contents, and being able to create them. How should this work?

2.4.1 Root Node

The root node is the root of the tree. It does not occur anywhere else in the tree. It has a single child which is the element node for the document element of the document.

2.4.2 Element Nodes

There is an element node for every element in the document. An element has an expanded name consisting of a local name and a possibly null URI (see [W3C XML Names]); the URI will be null if the element type name has no prefix and there is no default namespace in scope.

An element node also has a *namespace prefix map* that specifies the namespace URI for each namespace prefix that is in scope for the element. The semantics of a document type may treat parts of attribute values or data content as namespace prefixes. The namespace prefix map ensures that the semantics can be preserved when the tree is written out as XML. When writing an element node out as XML, an XSL processor must add sufficient namespace-declaring attributes to the start-tag to ensure that all prefixes in the element node's namespace prefix map are correctly declared.

The children of an element node are the element nodes and characters for its content. Entity references to both internal and external entities are expanded. Character references are resolved.

The *descendants* of an element node are the character children, the element node children, and the descendants of the element node children.

The set of all element nodes in a document can be ordered according to the order of the start-tags of the elements in the document; this is known as *document order*.

2.4.2.1 Unique IDs

An element object may have a unique identifier (ID). This is the value of the attribute which is declared in the DTD as type `ID`. Since XSL must also work with XML documents that do not have a DTD, stylesheets may specify which attributes in the source document should be treated as IDs. The `xsl:id` element has a required `attribute` attribute, which gives the name of an attribute in the source document that should be treated as specifying the element's ID. A stylesheet may contain more than one `xsl:id` element, for cases where the source document uses several attributes as IDs. An `xsl:id` element also has an optional `element` attribute which specifies the name of an element type; when the `element` attribute is specified, then the `xsl:id` element specifies that the `attribute` attribute of `element` elements are treated as IDs. `xsl:id` elements may only occur in the stylesheet body (not within a rule). The following causes XSL to treat all `name` attributes in the source document as IDs.

```
<xsl:id attribute="name"/>
```

It is an error if, as a consequence of the use of `xsl:id`, there is more than one element with the same ID in the source tree. An XSL processor may signal the error; if it does not signal the error, it must recover by treating only the first (in document order) of the elements as having that ID.

2.4.2.2 Base URI

An element 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 occurs in an external entity, the base URI of that element is the URI of the external entity. Otherwise the base URI is the base URI of the document.

2.4.3 Attribute Nodes

Each element node has an associated set of attribute nodes. A defaulted attribute is treated the same as a specified attribute. If an attribute was declared for the element type, but the default was declared as `#IMPLIED`, and the attribute was not specified on the element, then the element’s attribute set does not contain a node for the attribute.

An attribute node has an expanded name and has a string value. The expanded name consists of a local name and a possibly null URI (see [W3C XML Names]); the URI will be null if the specified attribute name did not have a prefix. The value is the normalized value as specified by the XML Recommendation [W3C XML]. An attribute value whose value is of zero length is not treated specially.

There are no attribute nodes for attributes that declare namespaces (see [W3C XML Names]).

> **Issue (external-dtd):** Should we specify something about how we expect XSL processors to process external DTDs and parameter entities? For example, what happens if an attribute default is declared in an external DTD?

2.4.4 Character Data

Each character within a CDATA section is treated as character data. Thus `<![CDATA[<]]>` in the source document will treated the same as `<`. Characters inside comments or processing instructions are not character data. Line-endings in external entities are normalized to #xA as specified in the XML Recommendation [W3C XML].

2.4.5 Whitespace Stripping

After the tree has been constructed, but before it is otherwise processed by XSL, some whitespace characters may be stripped. The stripping process takes as input a set of element types for which whitespace must be preserved. The stripping process is applied to both stylesheets and source documents, but the set of whitespace-preserving element types is determined differently for stylesheets and for source documents.

A character object is preserved if any of the following apply:

- The element type of the parent of the character is in the set of whitespace-preserving element types.
- It is part of a chunk that contains at least one non-whitespace character. As in XML, a whitespace character is #x20, #x9, #xD or #xA. A chunk of characters is a maximal sequence of sibling characters without any intervening elements.
- An ancestor element of the character 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 character object 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 types consists of just `xsl:text`.

For source documents, the set of whitespace-preserving element types is determined using the stylesheet as follows:

- If the `xsl:stylesheet` element specifies a `default-space` attribute with a value of `strip`, then the set is initially empty. Otherwise the set initially contains all element types that occur in the document.
- The `xsl:strip-space` element causes an element type to be removed from the set of whitespace-preserving element types. The `element` attribute gives the name of the element type.
- The `xsl:preserve-space` element causes an element type to be added to the set whitespace-preserving element types. The `element` attribute gives the name of the element type.

> **Issue (declare-multiple-elements):** Should the value of the `element` attribute of `xsl:strip-space`, `xsl:preserve-space` and `xsl:id` be a list of element type names (and thus be renamed to `elements`)? If so, should the `attribute` attribute of `xsl:id` also be a list of attribute names?

> **Ed. Note:** Clarify how these declarations interact with each other and with xsl:import.

The `xsl:stylesheet` element can include an `indent-result` attribute with values `yes` or `no`. If the stylesheet specifies `indent-result="yes"`, then the XSL processor may add whitespace to the result tree (possibly based on whitespace stripped from either the source document or the stylesheet) in order to indent the result nicely; if `indent-result="no"`, it must not add any whitespace to the result. When adding whitespace with `indent-result="yes"`, the XSL processor can use any algorithm provided that the result is the same as the result with `indent-result="no"` after whitespace is stripped from both using the process described with the set of whitespace-preserving element types consisting of just `xsl:text`.

2.5 Template Rules

A template rule is specified with the `xsl:template` element. The `match` attribute identifies the source node or nodes to which the rule applies. The content of the `xsl:template` element is the template.

For example, an XML document might contain:

```
This is an <emph>important</emph> point.
```

The following template rule matches elements of type `emph` and has a template which produces a `fo:sequence` formatting object with a `font-weight` property of `bold`.

```
<xsl:template match="emph">
  <fo:sequence font-weight="bold">
    <xsl:process-children/>
  </fo:sequence>
</xsl:template>
```

As described later, the `xsl:process-children` element recursively processes the children of the source element.

2.5.1 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 are less important than the most important matching template rule or rules are eliminated from consideration.

2. Next, all matching template rules that are less specific (as defined in **Section 2.6.11: Specificity**) than the most specific matching template rule or rules are eliminated from consideration.

3. Next, all matching template rules that have a lower priority than the matching template rule or rules with the highest priority are eliminated from consideration. The priority of a rule is specified by the `priority` attribute on the rule. The value of this must be an integer (positive or negative). The default priority is 0. The integer must not be greater than 2147483647 nor less than -2147483648.

It is an error if this leaves more than one matching template rule. An XSL 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.

2.5.2 Built-in Template Rule

There is a built-in template rule to allow recursive processing to continue in the absence of a successful pattern match by an explicit rule in the stylesheet. This 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:process-children/>
</xsl:template>
```

The built-in template rule is treated as if it were imported implicitly before the stylesheet and so is considered less important than all other template rules. Thus the author can override the built-in rule by including an explicit rule with `match="*|/"`.

2.6 Patterns

A pattern is a string which is matched against an element in the source document. The most common pattern specifies the element type name of a matching element. For example, the pattern `emph` matches an element whose type is `emph`. More complex patterns specify the element types of ancestors of a matching element. For example, the pattern `olist/item` matches an element whose type is `item` with a parent element of type `olist`. Each element type in the list of ancestors may be followed by a list of "qualifiers" separated by commas. For example,

```
list[attribute(ordered)="yes")]/item[first-of-type()]
```

matches an element whose type is `item`, which is the first amongst its siblings of this type, and which has a parent of type `list` with an `ordered` attribute equal to `yes`. This section describes the syntax and semantics of patterns in detail.

A pattern that is matched against an element is known as a *match pattern*. Patterns in `xsl:template` are match patterns.

A pattern can also be used to select a list of nodes; a pattern that is used for this is known as a *select pattern*. In a select pattern, there is a current node which provides a context for the selection. The pattern selects the list of the source nodes that match the pattern. Nodes in the selected list are in document order. Patterns in `xsl:process`, `xsl:for-each` and `xsl:value-of` are select patterns.

In the following grammar, the nonterminals *NCName* and *QName* are defined in [W3C XML Names], and *S* is defined in [W3C XML].

> **Issue (pattern-text-content):** Should patterns be able to match elements based on their textual content? If XSL is not to discriminate between DTDs that use attributes and those that use elements, then it needs at least a qualifier that matches an element whose pcdata content is exactly some specified string.
>
> **Issue (pattern-text-target):** At the moment, the only kind of node that can match a pattern is an element. Should it be possible to have a pattern that matches text? What would this mean for the processing model?
>
> **Issue (pattern-pi-target):** Should it be possible to have a pattern that matches a processing instruction?

Issue (regex): Should XSL support regular expressions for matching against any or all of pcdata content, attribute values, attribute names, element type names?

2.6.1 Alternative Patterns

A pattern may consist of a set of patterns representing ancestry separated by the `|` character. This indicates that an element matching any one of the ancestry patterns matches the entire pattern. Each ancestry pattern may itself have the capability of a full pattern except for the `|` operator.

Alternation

[1]	Pattern	::=	OrPattern
[2]	OrPattern	::=	AncestryPattern ('|' AncestryPattern)*

This example creates an `fo:sequence` for either `emph` or `strong` elements:

```
<xsl:template match="emph | strong">
  <fo:sequence font-weight="bold">
    <xsl:process-children/>
  </fo:sequence>
</xsl:template>
```

2.6.2 Matching on Element Ancestry

Element ancestry can be represented within the pattern by using the parent operator (`/`). This operator is based on the familiar directory navigation metaphor. Two patterns separated by the parent operator match an element if the right-hand side matches the element and the left-hand side matches the parent of the element.

Ancestry

[3]	AncestryPattern	::=	NodePatterns
			| (Anchor (AncestryOp NodePatterns)?)
			| (RootPattern NodePatterns?)
[4]	NodePatterns	::=	(ElementPatterns (AncestryOp AttributePattern)?)
			| AttributePattern
[5]	ElementPatterns	::=	ElementPattern (AncestryOp ElementPattern)*
[6]	AncestryOp	::=	'/' | '//'
[7]	ElementPattern	::=	ElementTypePattern ElementQualification?

For example, the following pattern matches `title` source elements that have a `section` element as a parent and a `chapter` element as a grandparent:

```
<xsl:template match="chapter/section/title">
  ...
</xsl:template>
```

While the parent operator specifies a parent-child relationship, the ancestor operator (`//`) specifies an ancestor-descendant relationship. Two patterns separated by the ancestor operator match an element if the right-hand side matches the element and the element has at least one ancestor that the left-hand side matches. Thus zero or more levels of hierarchy may intervene between the element matching the pattern specified on the left-hand side of the ancestor operator and the element matching the pattern specified on the right-hand side.

This example applies to `changed` elements which have a `para` element as an ancestor.

```
<xsl:template match="para//changed">
  ...
</xsl:template>
```

2.6.3 Anchors

The first component of an *AncestryPattern* can be an *Anchor*. An *Anchor* is a pattern that only a specific element in the source tree matches (if any elements at all match). The *Anchor* is said to *address* the specific element that matches it.

There are two kinds of anchor, relative and absolute. A relative anchor addresses an element relative to the current node. An absolute anchor addresses an element independently of any current node. A match pattern must not contain a *RelativeAnchor*; all anchors in a match pattern must be absolute. A select pattern may contain both kinds of anchor.

Anchors

[8]	Anchor	::=	AbsoluteAnchor | RelativeAnchor
[9]	AbsoluteAnchor	::=	IdAnchor
[10]	RelativeAnchor	::=	CurrentNodeAnchor | (ParentAnchor ('/' ParentAnchor)*) | AncestorAnchor
[11]	CurrentNodeAnchor	::=	'.'
[12]	ParentAnchor	::=	'..'

A *CurrentNodeAnchor* addresses the current node.

A select pattern is implicitly anchored to the current node: if an *AncestryPattern* in the *Pattern* does not start with an *Anchor* or a *RootPattern*, it is treated as if it were `./`*AncestryPattern*. For example, a select pattern of `foo` will be treated as `./foo` and will thus select the `foo` children of the current node.

A *ParentAnchor* addresses the parent of the current node.

2.6.3.1 Ancestor Anchors

Ancestor Anchors

[13]	AncestorAnchor	::=	'ancestor' '(' Pattern ')'

An *AncestorAnchor* addresses the nearest ancestor of the current node that matches the specified pattern. The pattern in a *AncestorAnchor* is a match pattern.

2.6.3.2 Id Anchors

An *IdAnchor* addresses the element whose ID (see **Section 2.4.2.1: Unique IDs**) is the specified *NCName*.

Id Anchors

[14]	IdAnchor	::=	'id' '(' NCName ')'

2.6.4 Matching the Root Node

A *RootPattern* matches the root node (see **Section 2.4.1: Root Node**).

Root Patterns

[15]	RootPattern	::=	'/'

2.6.5 Matching on Element Types

The simplest pattern consists of just an element type name. This matches any element of that type.

```
<xsl:template match="first-name">
  ...
</xsl:template>
```

The `*` pattern is a wildcard that matches a single element of any type. When used within an ancestry chain, the wildcard matches exactly one level of hierarchy.

Element Type Pattern

[16]	ElementTypePattern	::=	OneElementTypePattern | AnyElementTypePattern
[17]	OneElementTypePattern	::=	ElementTypeName
[18]	ElementTypeName	::=	QName
[19]	AnyElementTypePattern	::=	'*'

The following pattern matches any element that is an immediate child of a `data-samples` element.

```
<xsl:template match="data-samples/*">
  ...
</xsl:template>
```

When determining whether a source element matches an *ElementTypeName*, the expanded element type names are compared (see **Section 2.4.2: Element Nodes**).

> **Issue (pattern-namespace-wildcards):** Should patterns of the form `foo:*` or `*:foo` be allowed? If so, should `*` match any element or any element without a namespace URI?

2.6.6 Qualifiers

An element within the pattern hierarchy may have qualifiers applied to it, which further constrain which elements match the term. These qualifiers may constrain the element to have certain attributes or sub-elements or may constrain its position with respect to its siblings. The qualifiers are specified in square brackets (`[]`) following the element type name or wildcard symbol. A pattern matches only if all of the qualifiers are satisfied.

Qualifiers

[20]	ElementQualification	::=	'[' Qualifiers? ']'
[21]	Qualifiers	::=	Qualifier (',' Qualifier)*
[22]	Qualifier	::=	ChildQualifier | AttributeQualifier | PositionalQualifier

This example matches `author` elements within `book` elements where the book contains at least one `excerpt` sub-element and the author has a `degree` attribute:

```
<xsl:template match="book[excerpt]/author[attribute(degree)]">
  ...
</xsl:template>
```

> **Issue (sibling-qual):** Should there be qualifiers that constrain an element to have an immediately preceding or following sibling of a particular type?

2.6.7 Matching on Children

An element can be constrained to have a child element of a particular type by specifying the name of that type as a qualifier.

Child Qualifier

[23]	ChildQualifier	::=	ElementTypeName

This example has a pattern that matches `author` elements within `book` elements which also have `excerpt` children (the `author` and `excerpt` elements are siblings).

```
<xsl:template match="book[excerpt]/author">
  ...
</xsl:template>
```

> **NOTE:** There is no requirement that each child qualifier is matched by a distinct element. Thus `foo[bar,bar]` is matched by <foo><bar/></foo>.

2.6.8 Matching on Attributes

Attributes on the source element or any of its ancestor elements can also be used to determine whether a particular rule applies to an element. An attribute qualifier constrains an element either to have a specific attribute with a specific value, or to have a specific attribute with any value.

Attribute Qualifier

<table>
<tr><td>[24]</td><td>AttributeQualifier</td><td>::=</td><td>AttributePattern (’=’ AttributeValue)?</td></tr>
<tr><td>[25]</td><td>AttributePattern</td><td>::=</td><td>’attribute’ ’(’ QName ’)’</td></tr>
<tr><td>[26]</td><td>AttributeValue</td><td>::=</td><td>’"’ [^"]* ’"’
| "’" [^’]* "’"</td></tr>
</table>

When matching attribute names, the expanded names are compared (see **Section 2.4.3: Attribute Nodes**).

> **Issue (attribute-qual-case):** Do we need to be able to match attributes in a case insensitive way?

The following example matches an `item` element that has for its parent a `list` element which has a `compact` attribute:

```
<xsl:template match="list[attribute(compact)]/item">
  ...
</xsl:template>
```

The following example matches an `item` element that has for its parent a `list` element whose `liststyle` attribute has the value `enum`:

```
<xsl:template match="list[attribute(liststyle)='enum']/item">
  ...
</xsl:template>
```

It is also possible to select attribute nodes. Within a select pattern, an *AncestryPattern* can end with an *AttributePattern*. This will select the attribute node with the specified name for each element in the node list selected by the pattern preceding *AttributePattern*. Within a match pattern, an *AncestryPattern* must not end with an *AttributePattern*.

> **Issue (class-attribute):** Should there be a way of specifying that an attribute serves as a class attribute and then pattern syntax that treats class attributes specially?

2.6.9 Matching on Position

Positional qualifiers may be used to further constrain the pattern to match on the element's position or uniqueness among its siblings.

XSL defines the following positional qualifiers:

- `first-of-type()`. The element must be the first sibling of its type.
- `not-first-of-type()`. The element must not be the first sibling of its type.
- `last-of-type()`. The element must be the last sibling of its type.
- `not-last-of-type()`. The element must not be the last sibling of its type.
- `first-of-any()`. The element must be the first sibling element of any type.
- `not-first-of-any()`. The element must not be the first sibling element of any type.
- `last-of-any()`. The element must be the last sibling element of any type.
- `not-last-of-any()`. The element must not be the last sibling element of any type.
- `only-of-type()`. The element must have no element siblings of the same type.
- `not-only-of-type()`. The element must have one or more element siblings of the same type.
- `only-of-any()`. The element must have no element siblings at all.
- `not-only-of-any()`. The element must have one or more element siblings.

Positional Qualifier

---

[27]	PositionalQualifier	::=	Position '(' ')'
[28]	Position	::=	'first-of-type' | 'not-first-of-type' | 'first-of-any' | 'not-first-of-any' | 'last-of-type' | 'not-last-of-type' | 'last-of-any' | 'not-last-of-any' | 'only-of-type' | 'not-only-of-type' | 'only-of-any' | 'not-only-of-any'

The following pattern matches the first `item` in a `list`:

```
<xsl:template match="list/item[first-of-type()]">
  ...
</xsl:template>
```

The following rule is used for `appendix` elements when there is only one appendix:

```
<xsl:template match="backmatter/appendix[only-of-type()]">
  ...
</xsl:template>
```

2.6.10 Whitespace in Patterns

For readability, whitespace may be used in patterns even though not explicitly allowed by the grammar: *PatternWhitespace* may be freely added within patterns before or after any *PatternToken*.

Pattern Lexical Structure

[29]	PatternToken	::=	'/' | '//' | '(' | ')' | '|' | '[' | ']' | ',' | '=' | '.' | '..' | '*' | 'attribute' | 'id' | 'ancestor' | NCName | QName | AttributeValue | Position
[30]	PatternWhitespace	::=	S

2.6.11 Specificity

When a source element is matched against multiple patterns, it is possible for it to match more than one distinct pattern. In this situation, XSL defines which pattern or patterns are the most specific.

A pattern that starts with an *IdAnchor* is more specific than a pattern that does not. If two patterns both start with an *IdAnchor* or both do not start with an *IdAnchor*, then the one with the more components is the more specific, where a component is either a *Qualifier* or a *OneElementTypePattern*.

For example, the following patterns are in decreasing order of specificity:

1. `id(employee-of-the-month)`
2. `employee[attribute(type)='contract',attribute(country)='USA']`
3. `employee[attribute(type)='contract']`
4. `employee`
5. `*`

When a pattern contains alternatives separated by `|`, each alternative is treated separately for specificity purposes. A rule that contains a pattern with two alternatives is equivalent to two rules with the same content each specifying one of the alternatives as its patterns. For example

```
<xsl:template match="EMPH|B" priority="1">
  <fo:sequence font-weight="bold"><xsl:process-children/></fo:sequence>
</xsl:template>
```

is equivalent to

```
<xsl:template match="EMPH" priority="1">
  <fo:sequence font-weight="bold"><xsl:process-children/></fo:sequence>
</xsl:template>
<xsl:template match="B" priority="1">
  <fo:sequence font-weight="bold"><xsl:process-children/></fo:sequence>
</xsl:template>
```

2.7 Templates

2.7.1 Overview

When the rule that is to be applied to the source element has been identified, the rule's template is instantiated. A template can contain literal result elements, character data and instructions for creating fragments of the result tree. Instructions are represented by elements in the XSL namespace.

Instructions can select descendant elements for processing. There are two such instructions, the `xsl:process-children` instruction and the `xsl:process` instruction; the `xsl:process-children` instruction processes the immediate children of the source element, while the `xsl:process` instruction processes elements selected by a specified pattern.

```
<xsl:template match="chapter/title">
  <fo:rule-graphic/>
  <fo:block space-before="2pt">
    <xsl:text>Chapter </xsl:text>
    <xsl:number/>
    <xsl:text>: </xsl:text>
    <xsl:process-children/>
  </fo:block>
  <fo:rule-graphic/>
</xsl:template>
```

Note that `xsl:process-children` selects any character children of the source element as well as the element children. The result in this case is the sequence of results of processing the individual character children and element children in sequence.

> **Issue (instruction-next-match):** Should we add an instruction with the functionality of `next-match` in DSSSL?

> **Issue (instruction-error):** Should there be an instruction that generates an error, like the `error` procedure in DSSSL?

> **Issue (instruction-sort):** There needs to be an instruction that can do sorting. How should this work?

> **Issue (instruction-result-number):** There needs to be instruction that allows result elements to be numbered. How should this work?

> **Issue (modes):** How should the functionality of DSSSL modes be provided? Should nested template rules be used for this? Is there a way of getting source elements to appear in multiple places in the output that can be implemented in a single pass?

> **Issue (identity-transform):** There needs to be a way to do the identity transformation (creating a result tree identical to the source tree). How should this be done?

2.7.2 Literal Result Elements

In a template an element in the stylesheet that does not belong to the XSL namespace is instantiated to create an element node of the same type; the created element node will have the attributes that were specified on the element in the template tree.

The value of an attribute of a literal result element is interpreted as an attribute value template: it can contain string expressions contained in curly braces (`{}`).

The namespace prefix map of the result element node is the namespace prefix map of the element node in the stylesheet after the removal of any prefixes that map to the XSL namespace URI.

Since an XSL processor acts on elements belonging to the XSL namespace, the problem arises of how to create elements belonging to the XSL namespace. A namespace whose URI is `http://www.w3.org/TR/WD-xsl` followed by one or more occurrences of `/quote` is called a quoted namespace. Quoted namespaces are treated specially: before a result tree is written out as XML all quoted namespace URIs in expanded names and in namespace prefix maps are unquoted by removing the final `/quote`.

2.7.3 Named Attribute Sets

The `xsl:define-attribute-set` element defines a named set of attributes. The `name` attribute specifies the name of the attribute set. The content of the `xsl:define-attribute-set` element is an `xsl:attribute-set` element that specifies the attributes. A literal result element or an `xsl:attribute-set` element can specify an attribute set name as the value of the `xsl:use` attribute.

The following example creates a named attribute set `title-style` and uses it in a template rule.

```
<xsl:define-attribute-set name="title-style">
  <xsl:attribute-set font-size="12pt"
                     font-weight="bold"/>
</xsl:define-attribute-set>

<xsl:template match="chapter/heading">
  <fo:block xsl:use="title-style" quadding="start">
    <xsl:process-children/>
  </fo:block>
</xsl:template>
```

If the `xsl:use` attribute is specified on an element that also specifies a value for an attribute that is also part of the attribute set named by `xsl:use`, the attribute in the named attribute set is not used.

Multiple definitions of an attribute set with the same name are merged. An attribute from a definition that is more important takes precedence over an attribute from a definition that is less important. It is an error if there are two attribute sets with the same name that are equally important and that both contain the same attribute unless there is a more important definition of the attribute set that also contains the attribute. An XSL processor may signal the error; if it does not signal the error, it must recover by choosing from amongst the most important definitions that specify the attribute the one that was specified last in the stylesheet.

An `xsl:use` attribute may specify a list of attribute set names separated by whitespace. These attribute sets will be merged treating the list as being in order of increasing importance.

2.7.4 Literal Text in Templates

A template can also contain PCDATA. Each data character in a template remaining after whitespace has been stripped as specified in **Section 2.4.5: Whitespace Stripping** will create a data character in the result tree.

Literal data characters may also be wrapped in an `xsl:text` element. This wrapping may change what whitespace characters are stripped (see **Section 2.4.5: Whitespace Stripping**) but does not affect how the characters are handled by the XSL processor thereafter.

2.7.5 Processing with `xsl:process-children`

> **Ed. Note:** There is no WG consensus on the name xsl:process-children.
>
> **Ed. Note:** When we add to select patterns the ability to select characters as well as elements, the functionality of xsl:process-children will be available from xsl:process.

This example creates a block for a `chapter` element and then processes its immediate children.

```
<xsl:template match="chapter">
  <fo:block>
    <xsl:process-children/>
  </fo:block>
</xsl:template>
```

The xsl:process-children instruction processes all of the children of the current node, including characters. However, characters that have been stripped as specified in **Section 2.4.5: Whitespace Stripping** will not be processed.

Processing a character in the source tree adds the character to the result tree. Note that this works at the tree level. Thus, markup of < in content will be represented by a character < in the source tree which will, with the built-in template rules, turn into a < character in the result tree, which would be represented by the markup < (or an equivalent character reference) when the result tree is externalized as an XML document.

2.7.6 Processing with xsl:process

> **Ed. Note:** There is no WG consensus on the name xsl:process.
>
> **Issue (process-name):** What should xsl:process be called?

The xsl:process element processes elements selected by a pattern. The pattern of an xsl:process element is a select pattern and so is implicitly anchored to the current node. The following example processes all of the author children of the author-group:

```
<xsl:template match="author-group">
  <fo:sequence>
    <xsl:process select="author"/>
  </fo:sequence>
</xsl:template>
```

The xsl:process element processes all *elements* which match the specified pattern. Character data content is not matched by an xsl:process element. The pattern must not contain an *AttributePattern* except as part of an *AttributeQualifier*

The pattern controls the depth at which matches occur. The following example processes all of the first-names of the authors that are direct children of author-group:

```
<xsl:template match="author-group">
  <fo:sequence>
    <xsl:process select="author/first-name"/>
  </fo:sequence>
</xsl:template>
```

The // operator can be used in the pattern to allow the matches to occur at arbitrary depths.

This example processes all of the heading elements contained in the book element.

```
<xsl:template match="book">
  <fo:block>
    <xsl:process select=".//heading"/>
  </fo:block>
</xsl:template>
```

An *AncestorAnchor* in the pattern allows the processing of elements that are not descendants of the current node. This example finds an employee's department and then processes the group children of the `department`.

```
<xsl:template match="employee">
  <fo:block>
    Employee <xsl:process select="name"/> belongs to group
    <xsl:process select="ancestor(department)/group"/>
  </fo:block>
</xsl:template>
```

This example assumes that a `department` element contains `group` and `employee` elements (at some level). When processing the `employee` elements, the *AncestorAnchor* in the pattern allows navigation upward to the `department` element in order to extract the information about the group to which the employee belongs.

An *IdAnchor* allows processing of elements with a specific ID. For example, this template rule applies to elements with the ID `cfo`; the second `xsl:process` element processes the `name` child of the element with ID `ceo`:

```
<xsl:template match="id(cfo)">
  <xsl:process select="name"/> reports to <xsl:process select="id(ceo)/name"/>
</xsl:template>
```

Multiple `xsl:process` 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:process select="sales/domestic"/>
  </TABLE>
  <TABLE>
    <xsl:process 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> </doc>
> ```
>
> the rule
>
> ```
> <xsl:template match="doc">
>   <xsl:process select=".//div"/>
> </xsl:template>
> ```
>
> will process both the outer `div` and inner `div` elements.

Use of *Anchor*s in patterns in `xsl:process` can lead to infinite loops. It is an error if, during the invocation of a rule for an element, that same rule is invoked again for that element. An XSL processor may signal the error; if it does not signal the error, it must recover by creating an empty result tree structure for the nested invocation.

Issue (select-function): What mechanisms should be provided for selecting elements for processing? For example, how can elements specified indirectly be handled? Suppose there's an `xref` element with a `ref` attribute that specifies the ID of a `div` element. The template for `xref` needs to select `title` child of the `div` element referenced by the `ref` attribute. Should it be possible to select elements in other XML documents?

2.7.7 Direct Processing

When the result has a known regular structure, it is useful to be able to specify directly the template for selected elements. The `xsl:for-each` element contains a template which is instantiated for each element selected by the pattern specified by the `select` attribute.

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:process select="name"/>
          </TH>
          <xsl:for-each select="order">
        <TD>
          <xsl:process-children/>
        </TD>
          </xsl:for-each>
        </TR>
      </xsl:for-each>
    </TBODY>
      </TABLE>
    </BODY>
  </HTML>
</xsl:template>
```

As with `xsl:process` the pattern is a select pattern and so is implicitly anchored to the current node. The `select` attribute is required. The pattern must not contain an *AttributePattern* except as part of an *AttributeQualifier*.

2.7.8 Numbering in the Source Tree

The `xsl:number` element does numbering based on the position of the current node in the source tree.

The `xsl:number` element can have the following attributes:

- The `level` attribute specifies what levels of the source tree should be considered; it has the values `single`, `multi` or `any`. The default is `single`.
- The `count` attribute is a match pattern that specifies what elements should be counted at those levels. The `count` attribute defaults to the element type name of the current node.
- The `from` attribute is a match pattern that specifies where counting starts from.

In addition the `xsl:number` element has the attributes specified in **Section 2.7.9: Number to String Conversion Attributes** for number to string conversion.

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 nearest ancestor (including the current node as its own ancestor) 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.
- When `level="multi"`, 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 elements that match the `count` pattern; it then maps each element of the list to one plus the number of preceding siblings of that element 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.
- When `level="any"`, it constructs a list of length one containing one plus number of elements at any level of the document that start before this node and that match the `count` pattern. If the `from` attribute is specified, then only elements after the first element before this element that match the `from` pattern are considered.

The list of numbers is then converted into a string using the attributes specified in **Section 2.7.9: Number to String Conversion Attributes**; when used with `xsl:number` 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:process-children/>
  </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="multi"
                 count="chapter|section|subsection"
                 format="1.1. "/>
     <xsl:process-children/>
  </fo:block>
</xsl:template>

<xsl:template match="appendix//title">
  <fo:block>
     <xsl:number level="multi"
                 count="appendix|section|subsection"
                 format="A.1. "/>
     <xsl:process-children/>
  </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:process-children/>
  </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:process-children/>
 </fo:block>
</xsl:template>
```

2.7.9 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. The alphanumeric tokens (format tokens) specify the format to be used for each number in the list; the non-alphanumeric tokens (separator tokens) specify the separators used to join numbers in the list. Alphanumeric means any character that has a Unicode category of Nd, Nl, No, Lu, Ll, Lt, Lm or Lo. If the first token is a separator token, then the constructed string will start with that token; if the last token is a separator token, then the constructed string will end with that token. 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. The format token specifies the string to be used to represent the number 1. If there are more than n numbers, then the n-th number will be separated from the following number by the separator token following the n-th format token; if there is no such separator token, then the last separator token will be used; if there are no separator tokens, then `.` will be used.

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 2.0 character property database), and the Unicode value of preceding characters is one less than the Unicode value of the last character. This 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 vii ix x ....`

- A format token `I` generates the sequence `I II III IV V VI VII VII IX X ....`

- Any other format token indicates a numbering sequence that starts with that token. If an implementation does not support a numbering system that starts with that token, it must use a format token of `1`.

When numbering with an alphabetic sequence, the `xml:lang` attribute specifies which language's alphabet is to be used.

> **NOTE:** This can be considered as specifying the language of the value of the `format` attribute and hence is consistent with the semantics of `xml:lang`.

The `letter-value` attribute disambiguates between numbering schemes that use letters. In many languages there are two commonly used numbering schemes that use letters. One numbering scheme assigns numeric values to letters in alphabetic sequence, and the other assigns numeric values to each letter in some other manner. 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 `other` specifies the other sequence.

The `digit-group-sep` attribute gives the separator between groups of digits, and the optional `n-digits-per-group` specifies the number of digits per group. For example, `digit-group-sep=","` and `n-digits-per-group="3"` would produce numbers of the form `1,000,000`.

The `sequence-src` attribute gives the URI of a text resource that contains a whitespace separated list of the members of the numbering sequence.

> **Ed. Note:** Specify what should happen when the sequence runs out.

Here are some examples of conversion specifications:

- `format="ア"` specifies Katakana numbering
- `format="イ"` specifies Katakana numbering in the "iroha" order
- `format="๑"` specifies numbering with Thai digits
- `format="א" letter-value="other"` specifies "traditional" Hebrew numbering
- `format="ა" letter-value="other"` specifies Georgian numbering
- `format="α" letter-value="other"` specifies "classical" Greek numbering
- `format="а" letter-value="other"` specifies Old Slavic numbering

2.7.10 Conditionals within a Template

There are two instructions in XSL which 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.

2.7.10.1 Conditional Processing with `xsl:if`

The `xsl:if` element has a single attribute, `test` which specifies a select pattern. The content is a template. If the pattern selects a non-empty list of elements, then the content 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:process-children/>
  <xsl:if test=".[not-last-of-type()]">, </xsl:if>
</xsl:template>
```

2.7.10.2 Conditional Processing with `xsl:choose`

The `xsl:choose` element selects one among a number of possible alternatives. It consists of a series of `xsl:when` elements followed by an optional `xsl:otherwise` element. Each `xsl:when` element has a single attribute, `test`, which specifies a select pattern; the test is treated as true if the pattern selects a non-empty list of elements. 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. 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:choose>
        <xsl:when test='ancestor(orderedlist/orderedlist)'>
          <xsl:number format="i"/>
        </xsl:when>
        <xsl:when test='ancestor(orderedlist)'>
          <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:process-children/>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>
```

2.7.11 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 string constant. The `xsl:value-of` element does this with a string expression that is specified as the value of the `expr` attribute. String expressions can also be used inside attribute values of literal result elements by enclosing the string expression in curly brace (`{}`).

2.7.11.1 String Expressions

String Expressions

[31]	StringExpr	::=	Pattern | ConstantRef | MacroArgRef

The value of a string expression that is a pattern is the character content of the first node selected by the pattern. If the first node is an attribute node, the value is the attribute value. If the first node is an element node, the value is computed by concatenating all characters that are descendants of the element node in the order in which they occur in the document. The pattern is a select pattern and so is implicitly anchored to the current node.

> **Issue (resolve-expr):** Do we need a `resolve(pattern)` string expression that treats the characters as a relative URI and turns it into an absolute URI using the base URI of the addressed node?

2.7.11.2 Using String Expressions with `xsl:value-of`

The `xsl:value-of` element is replaced by the value of the string expression specified by the `expr` attribute. The `expr` attribute is required.

For example, the following creates an HTML paragraph from a `person` element with `first-name` and `surname` attributes.

```
<xsl:template match="person">
  <P>
   <xsl:value-of expr="attribute(first-name)"/>
   <xsl:text> </xsl:text>
   <xsl:value-of expr="attribute(surname)"/>
  </P>
</xsl:template>
```

For example, the following creates an HTML paragraph from a `person` element with `first-name` and `surname` children elements.

```
<xsl:template match="person">
  <P>
   <xsl:value-of expr="first-name"/>
   <xsl:text> </xsl:text>
   <xsl:value-of expr="surname"/>
  </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 an ancestor element of the procedure. It also assumes that if more than one ancestor has a `security` attribute then the security level is determined by the closest such ancestor of the procedure.

```
<xsl:template match="procedure">
  <fo:block>
    <xsl:value-of expr="ancestor(*[attribute(security)])/attribute(security)"/>
  </fo:block>
  <xsl:process-children/>
</xsl:template>
```

> **Issue (inherited-attribute):** Unless an element counts as one of its own ancestors, using `ancestor(*[attribute(security)])/attribute(security)` won't work to get the inherited value of an attribute. We could either say `ancestor` always includes the current node; alternatively we could provide a variant of `ancestor` that does include the current node; alternatively we could provide a select pattern of the form `inherited-attribute(security)`.

2.7.11.3 Attribute Value Templates

In an attribute value that is interpreted as an *attribute value template*, such as an attribute of a literal result element, string expressions can be used by surrounding the string expression with curly braces (`{}`). The attribute value template is instantiated by replacing the string expression together with surrounding curly braces by the value of the string expression.

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` constant and the content 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 the `width` attribute of the `size` child of the `photograph` element:

```
<xsl:define-constant name="image-dir" value="/images"/>

<xsl:template match="photograph">
<IMG SRC="{constant(image-dir)}/{href}" WIDTH="{size/attribute(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 a string 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 a string expression without being followed by a second right curly brace; an XSL processor may signal the error or recover by treating the right curly brace as if it had been doubled. A right curly brace inside an *AttributeValue* in a string expression is not recognized as terminating the string expression.

Curly braces are *not* recognized recursively inside string expressions. For example:

```
<a href="#{id({attribute(ref)})/title}">
```

is *not* allowed.

2.7.12 String Constants

Global string constants may be defined using a `define-constant` element. The name attribute specifies the name of the constant, and the `value` attribute specified the value.

A stylesheet must not contain more than one definition of a constant with the same name and same importance. A definition of a constant will not be used if there is another definition of a constant with the same name and higher importance.

String constants are referenced using a *ConstantRef* string expression.

String Constant References

[32]	ConstantRef	::=	'constant' '(' NCName ')'

```
<xsl:define-constant name="para-font-size" value="12pt"/>

<xsl:template match="para">
 <fo:block font-size="{constant(para-font-size)}">
   <xsl:process-children/>
 </fo:block>
</xsl:template>
```

> **Issue (local-constants):** Should there be a way to define local constants?

The `value` attribute is interpreted as an attribute value template. If the value of a constant definition `x` references a constant `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 constant definitions because of dependency cycles.

2.7.13 Macros

> **Issue (macro-name):** Should macros be called something else?

Parts of templates can also be factored out of similar rules into macros for reuse. Macros allow authors to create aggregate result fragments and refer to the composite as if it were a single object. In this example, a macro is defined for a boxed paragraph with the word "Warning!" preceding the contents. The macro is referenced from a rule for `warning` elements.

```
<xsl:define-macro name="warning-para">
  <fo:box>
    <fo:block>
      <xsl:text>Warning! </xsl:text>
      <xsl:contents/>
    </fo:block>
  </fo:box>
</xsl:define-macro>

<xsl:template match="warning">
  <xsl:invoke macro="warning-para">
    <xsl:process-children/>
  </xsl-invoke>
</xsl:template>
```

Macros are defined using the `define-macro` element. The `name` attribute specifies the name of the macro being defined. The content of the `define-macro` element is a template, called the body of the macro. A macro is invoked using the `xsl:invoke` element; the content of `xsl:invoke` is a template. The name of the macro to be invoked is given by the `macro` attribute. Invoking a macro first instantiates the content of `xsl:invoke`. It then instantiates the body of the invoked macro passing it the result tree fragment created by the instantiation of the content of `xsl:invoke`; this fragment can be inserted in the body of the macro using the `xsl:contents` element.

Macros allow named arguments to be declared with the `xsl:macro-arg` element; the `name` attribute specifies the argument name, and the optional `default` attribute specifies the default value for the argument. Within the body of a macro, macro arguments are referenced using a *MacroArgRef* string expression. It is an error to refer to a macro argument that has not been declared. An XSL processor may signal the error; if it does not signal the error, it must recover by using an empty string. Arguments are supplied to a macro invocation using the code `xsl:arg` element; the `name` attribute specifies the argument name, and the `value` attribute specifies the argument value. It is an error to supply an argument to a macro invocation if the macro did not declare an argument of that name. An XSL processor may signal the error; if it does not signal the error, it must recover by ignoring the argument. The `value` attribute of `xsl:arg` and the `default` attribute of `xsl:macro-arg` are interpreted as attribute value templates; they can contain string expressions in curly braces as with literal result elements.

Macro Argument References

[33]	MacroArgRef	::=	'arg' '(' NCName ')'

This example defines a macro for a `numbered-block` with an argument to control the format of the number.

```
<xsl:define-macro name="numbered-block">
  <xsl:macro-arg name="format" default="1. "/>
  <xsl:number format="{arg(format)}"/>
  <fo:block/>
    <xsl:contents/>
  </fo:block>
</xsl:define-macro>

<xsl:template match="appendix/title">
  <xsl:invoke name="numbered-block">
    <xsl:arg name="format" value="A. "/>
    <xsl:process-children/>
  </xsl:invoke>
</xsl:template>
```

It is an error if a stylesheet contains more than one definition of a macro with the same name and same importance. An XSL processor may signal the error; if it does not signal the error, if must recover by choosing from amongst the definitions with highest importance the one that occurs last in the stylesheet.

> **Issue (macro-arg-syntax):** The proposal used the same element for declaring macro arguments and for invoking them. Should these be separate elements and if so what should they be called?

2.8 Style Rules

This section will describe a facility similar to style rules in the XSL submisson.

> **Issue (style-rules):** How should style rules work?

2.9 Combining Stylesheets

XSL provides two mechanisms to combine stylesheets:

- an import mechanism that allows stylesheets to override each other, and
- an inclusion mechanism that allows stylesheets to be textually combined.

2.9.1 Stylesheet Import

An XSL stylesheet may contain `xsl:import` elements. All the `xsl:import` elements must occur at the beginning of the stylesheet. The `xsl:import` element has an `href` attribute whose value is the URI of a stylesheet to be imported. A relative URI is resolved relative to the base URI of the `xsl:import` element (see **Section 2.4.2.2: Base URI**).

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:import href="article.xsl"/>
  <xsl:import href="bigfont.xsl"/>
  <xsl:define-attribute-set name="note-style">
    <xsl:attribute-set font-posture="italic"/>
  </xsl:define-attribute-set>
</xsl:stylesheet>
```

Rules and definitions in the importing stylesheet are defined to be more *important* than rules and definitions in any imported stylesheets. Also rules and definitions in one imported stylesheet are defined to be more *important* than rules and definitions in previous imported stylesheets.

In general a more important rule or definition takes precedence over a less important rule or definition. This is defined in detail for each kind of rule and definition.

> **Issue (stylesheet-partition):** Should there be an XSL defined element that can be used to divide a stylesheet into parts, each of which is treated as if it were separately imported for precedence purposes?
>
> **Issue (import-source):** Provide a way for a stylesheet to import a stylesheet that is embedded in the document.
>
> **Issue (import-media):** Should we provide media-dependent imports as in CSS?
>
> **Ed. Note:** Say something about the case where the same stylesheet gets imported twice. This should be treated the same as importing a stylesheet with the same content but different URIs. What about import loops?

2.9.2 Stylesheet Inclusion

An XSL stylesheet may include another XSL stylesheet using an `xsl:include` element. The `xsl:include` element has an `href` attribute whose value is the URI of a stylesheet to be included. A relative URI is resolved relative to the base URI of the `xsl:include` element (see **Section 2.4.2.2: Base URI**). The `xsl:include` element can occur as the child of the `xsl:stylesheet` element at any point after all `xsl:import` elements.

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. Also 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. Unlike with `xsl:import`, the fact that rules or definitions are included does not affect the way they are processed.

> **Ed. Note:** What happens when a stylesheet directly or indirectly includes itself?

2.9.3 Embedding Stylesheets

Normally an XSL stylesheet is a complete XML document with the `xsl:stylesheet` element as the document element. However an XSL stylesheet may also be embedded in another resource. Two forms of embedding are possible:

- the XSL 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.

In the second case, the possibility arises of documents with inline style, that is documents that specify their own style. XSL does not define a specific mechanism for this. This is because this can be done by means of a general purpose mechanism for associating stylesheets with documents provided that:

- the mechanism allows a part of a resource to be specified as the stylesheet, for example by using a URI with a fragment identifier, and
- the mechanism can itself can be embedded in the document, for example as a processing instruction.

It is not in the scope of XSL to define such a mechanism.

> **NOTE:** This is because the mechanism should be independent of any one stylesheet mechanism.

The xsl:stylesheet element may have an ID attribute that specifies a unique identifier.

> **NOTE:** In order for such an attribute to be used with the `id` XPointer location term, it must actually be declared in the DTD as being an ID.

The following example shows how inline style can be accomplished using the `xml:stylesheet` processing instruction mechanism for associating a stylesheet with an XML document. The URI uses an XPointer in a fragment identifier to locate the `xsl:stylesheet` element.

```
<?xml version="1.0"?>
<?xml:stylesheet type="text/xsl" href="#id(style1)"?>
<!DOCTYPE doc SYSTEM "doc.dtd">
<doc>
<head>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" id="style1">
<xsl:import href="doc.xsl"/>
```

```
<xsl:template match="id(foo)">
 <fo:block font-weight="bold"><xsl:process-children/></fo:block>
</xsl:template>
</xsl:stylesheet>
</head>
<body>
<para id="foo">
...
</para>
</body>
</doc>
```

> **NOTE:** The `type` pseudo-attribute in the `xml:stylesheet` processing instruction identifies the stylesheet language, not the content type of the resource of which the stylesheet is a part.

2.10 Extensibility

This section will describe an extensibility mechanism for the tree construction process.

> **Issue (construct-extensibility):** Should there be some extensibility mechanism for the tree construction process? If so, how should it work? Should it be language independent?

3. Formatting Objects

This section describes the formatting vocabulary.

3.1 Introduction

The approach that we have taken in constructing this draft was to evaluate the requirements for print and online documents and established a target set of capabilities. This set of capabilities reflect the long-term goals of XSL

In this draft we concentrated on documenting a subset of the formatting capability that addressed basic WP-level pagination. We expect to cover more sophisticated pagination and support for layout-driven documents in later drafts.

3.2 Notations Used in this Section

The following typographic styles are used to identify different terms in this document:

`property-name`

> The name of a property or attribute. Always all lowercase and hyphenated between words.

`property-value`

> The value assigned to a property. Always all lowercase and hyphenated between words.
>
> This has varying forms dependent on the value type:

a ___-specifier

See defined types for the definitions of character-specifier, color-specifier, length-specifier, name-specifier, writing-mode-specifier.

keyword

The name of a value for enumerated types.

0.0pt

A measure (length-specifier) is always qualified with units.

0.0

A numeric value (integer and fraction).

1

An integer value.

()|[]{}...

Delimiters & operators, code fragments.

(*choice-1* | *choice-2* | ...)

A choice-list. Choose one of the values or options listed. These may be mixed (such as a choice between *none* and a *length-specifier*.

formatting-object-name

The name of a formatting object. Always all lowercase and hyphenated between words.

Formatting objects may be classified further as:

layout

Describes a master or layout.

flow

Classifies/groups content objects. Assigns formatting behaviors and properties.

lists

Flow objects that provide structure and behavior appropriate for ordered and unordered lists.

adornment

Flow objects that provide highlighting/borders for their children.

online

Objects that support interaction.

math

Components for the presentation of equations.

table

Parts of a table.

`Non-core`

Indicates that this object or property is not required for all XSL-compatible formatters. A fallback action will be defined if the property/object is not fully supported by an implementation.

defined-term

Terms found in the glossary in **Section 3.23: Defined Terms**.

3.3 Formatting Objects and Their Properties

The formatting objects and their properties are described in the following sections. This section outlines the plan for current and future drafts of this document.

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

Objects marked as **`non-core`** are not required to be implemented by all implementors of XSL.

The following formatting objects are defined in this draft. This list and the object definitions are subject to change in future drafts.

- Layout Formatting Objects
  - **`page-sequence`**

    Provides the mechanism to define master sequences and to associate content with those masters.

    > **Ed. Note:** In future drafts, this object will be split into a page-sequence object

and a flow-map object. This is not necessary for this draft because simple-page-master does not allow for flexible mapping.

- `simple-page-master`

  This object describes the general layout or layout sequencing for web pages (both print and online).

- Content Flow Objects

  - `queue`

    Gathers content to be placed in the page sequence.

  - `sequence`

    Groups content and allows the assignment of shared inherited properties.

  - `list`

    Groups all items in a list.

    - `list-item`

      Groups the list-item-label and list-item-body for each item in the list.

      - `list-item-label`

      Holds the number or label of a list item.

      - `list-item-body`

      Holds the main content of a list item. Allows for proper formatting of multi-paragraph list items.

  - `block`

    Used to represent paragraphs, titles, captions, etc. Allows formatting of text and graphics into textlines.

  - `character`

    Atomic unit to the formatter.

    Used when one needs to explicitly override a specific character or array of characters with a specific glyph for presentation.

    Ed. Note: Open issues: glyph specification & override, non-Unicode glyph selection, expert-set variants, ligatures.

- **`rule-graphic`**

  Rule-graphics are used to draw a graphic-line that is used to divide space on the page.

- **`graphic`**

  Holds an image or vector graphic.

  Placement in XSL may be inline or block-level.

  Content of the graphic may be instream or external (linked).

  > **Ed. Note:** Issue: Do we need to split this into 2 objects? Difference between unprocessible(unstylable) graphics (EPSF,GIF,TIFF...) and ones that may be parsed and processed (W3C-SVG?).

- **`score`**

  Highlights text. Used to produce underlines, strike-through, overbars, etc.

- Boxes

  Used to set backgrounds and borders.

  - **`inline-box`**

    Highlights text or graphics.

    Used to produce borders and backgrounds.

    Controls spacing surrounding the content.

  - **`block-level-box`**

    Highlights text or graphics.

    Used to produce borders and backgrounds.

    Controls spacing surrounding the content.

- Building Blocks

  Building-blocks are directives to the formatter to construct formatter-generated text object at this location in the content flow.

  - **`page-number`**

    Used to allow the formatter to produce page-numbers.

- Online Flow Objects

- **`link`**

  Web browser link.

- **`link-end-locator`**

  Target (destination) for a **`link`**.

3.4 Formatting Objects to be Defined in Subsequent Drafts

The following list identifies formatting functionality that we know to be missing from the current draft. Coverage of these areas will be added in the future. (Many of these areas require coordination with other W3C WGs.)

- Layout objects that deal with multi-column and more sophisticated page layouts.
- Content objects to support layout-driven formatting, side-by-side and floating objects, and extracted content (index, toc, endnotes, etc.)
- Additional international objects to handle mixed scripts, rubi, warichu, kumisuji and similar locale-specialized formatting.
- Additional building blocks to construct formatter-generated text, including auto-leaders, cross-references and other citations, layout-derived numbering, etc.
- Additional online objects
- Tables
- Math
- Others to be determined for online interaction and behavior

3.5 Page-sequence Layout Object

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.5.1 Purpose

This object describes the general layout or layout sequencing for web page (both print and online).

A **`page-sequence`** holds:

- a number of child **`simple-page-master`**s that define the layouts to be used for this sequence.

- a number of child **`queue`**s which hold the content to be placed in this sequence.

  Ed. Note: To support layout-driven documents in future drafts, the queues may not be held by the "page-sequence" and may be moved to a separate mapping object.

  NOTE: A document can contain multiple **`page-sequence`**s. For example, each chapter of a document could be a separate **`page-sequence`**; this would allow the chapter title within a header or footer.

3.5.2 Formatting Object Summary

<**`fo:page-sequence`**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**`fo:page-sequence`**>

3.5.3 Formatting Object's Formal Specification

A **`page-sequence`** holds:

- one or more child **`simple-page-master`**s that define the layouts to be used for this sequence
- one or more child **`queue`**s which hold the content to be placed in this sequence.

3.5.4 To Resolve

- Media selection:

  We have defined a master for scrolling and separate masters for paged presentations. In James' note on "Linking Stylesheets to XML Documents" he describes a simple mechanism to support media-driven selection of different stylesheets. We do not currently have a mechanism for supporting media switches within a stylesheet.

  Media selection interacts with sequence specification.

- Sequences:

  Ed. Note: A method to define the sequencing of page masters will be provided in a future draft.

3.6 Simple-page-master Layout Object

NOTE: Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.6.1 Purpose

A **`simple-page-master`** formatting object defines the layout of a page area. Masters may be repeated in accordance with the **`page-sequence`** specification.

The **`simple-page-master`** defines 5 areas for presentation within the page/window design (formatted area of the page). These are the header, body, footer, start-side, and end-side. It also provides a title, which has no properties defined in the **`simple-page-master`** object, but may for example be presented in a browser's title bar.

NOTE: The following **`simple-page-master`**s are the only ones supported in this draft of the standard:

first

A **`simple-page-master`** with `master-name=`*`first`*.

The master to be used for the first page in the sequence.

odd

A **`simple-page-master`** with `master-name=`*`odd`*.

The master to be used for odd-phased pages after the first page in the sequence.

even

A **`simple-page-master`** with `master-name=`*`even`*.

The master to be used for even-phased pages after the first page in the sequence.

For single-sided printing and paged online presentation, this master can be dropped.

scrolling

A **`simple-page-master`** with `master-name=`*`scrolling`*.

The master to be used for scrolling (non-paged) online presentation.

3.6.2 Formatting Object Summary

```
<fo:simple-page-master
```

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`master-name (DSSSL:-none-, CSS:-none-)` = name-specifier

Required

`background-attachment` = (*`scroll`* | *`fixed`*)

Inherited, Initial = *`scroll`*

`background-color` = a color-specifier or *`transparent`*

Inherited, Initial = *`transparent`*

`background-image` = a URI or *`none`*

Inherited, Initial = *`none`*

`background-position-x` = (*`a length-specifier`* | *`left`* | *`center`* | *`right`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`left`*

`background-position-y` = (*`a length-specifier`* | *`top`* | *`middle`* | *`bottom`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`top`*

`background-repeat` = (*`no-repeat`* | *`repeat`* | *`repeat-x`* | *`repeat-y`*)

Inherited, Initial = *`repeat`*

`page-height (DSSSL:-same-, CSS:height)` = length-specifier

Value(s): {*`auto`* | *`0..max-length`*}

Inherited, Initial = *`auto`*

`page-width (DSSSL:-same-, CSS:width)` = length-specifier

Value(s): {*`auto`* | *`0..max-length`*}

Inherited, Initial = *`auto`*

`page-writing-mode (DSSSL:-none-, CSS:-none-)` = writing-mode-specifier

Inherited, Initial = *`lr-tb`*

`margin-bottom (DSSSL:bottom-margin, CSS:margin-bottom)` = length-specifier

Value(s): { *`0..page-height`* }

Inherited, Initial = *`36.0pt`*

`margin-left (DSSSL:left-margin, margin-left)` = length-specifier

Value(s): { *`0..page-width`* }

Inherited, Initial = *`36.0pt`*

`margin-right (DSSSL:right-margin, CSS:margin-right)` = length-specifier

Value(s): { *`0..page-width`* }

Inherited, Initial = *`36.0pt`*

`margin-top (DSSSL:top-margin, CSS:margin-top)` = length-specifier

Value(s): { *`0..page-height`* }

Inherited, Initial = *`36.0pt`*

`body-overflow (DSSSL:-none-, CSS:overflow)` = (*`visible`* | *`hidden`* | *`scroll`* | *`auto`*)

Inherited, Initial = *`auto`*

`body-writing-mode (DSSSL:-none-, CSS:-none)` = a writing-mode-specifier | *`use-page-writing-mode`*

Inherited, Initial = *`use-page-writing-mode`*

`end-side-overflow (DSSSL:-none-, CSS:overflow)` = (*`visible`* | *`hidden`* | *`scroll`* | *`auto`*)

Inherited, Initial = *`auto`*

`end-side-separation (DSSSL:-none-, CSS:-none-)` = length-specifier

Value(s): { *`0`*.. available-size}

Inherited, Initial = *`0.0pt`*

`end-side-size (DSSSL:-none-, CSS:-none-)` = length-specifier

Value(s): {*0..page-height*}

Inherited, Initial = *0.0pt*

`end-side-writing-mode (DSSSL:-none-, CSS:-none-)` = a writing-mode-specifier | *use-page-writing-mode*

Inherited, Initial = *use-page-writing-mode*

`footer-overflow (DSSSL:-none-, CSS:overflow)` = (*visible* | *hidden* | *scroll* | *auto*)

Inherited, Initial = *auto*

`footer-precedence (DSSSL:-none-, CSS:-none)` = (*true* | *false*)

Inherited, Initial = *true*

`footer-separation (DSSSL:footer-margin, CSS:-none-)` = length-specifier

Value(s): {*0*.. available-size}

Inherited, Initial = *18.0pt*

`footer-size (DSSSL:-none-, CSS:-none-)` = length-specifier

Value(s): {*0..page-height*}

Inherited, Initial = *36.0pt*

`footer-writing-mode (DSSSL:-none-, CSS:-none)` = a writing-mode-specifier | *use-page-writing-mode*

Inherited, Initial = *use-page-writing-mode*

`header-overflow (DSSSL:-none-, CSS:overflow)` = (*visible* | *hidden* | *scroll* | *auto*)

Inherited, Initial = *auto*

`header-precedence (DSSSL:-none-, CSS:-none)` = (*true* | *false*)

Inherited, Initial = *true*

`header-separation (DSSSL:header-margin, CSS:-none-)` = length-specifier

Value(s): {*0*.. available-size}

Inherited, Initial = *18.0pt*

`header-size (DSSSL:-none-, CSS:-none-)` = length-specifier

Value(s): {*0..page-height*}

Inherited, Initial = *36.0pt*

`header-writing-mode (DSSSL:-none-, CSS:-none)` = a writing-mode-specifier | *use-page-writing-mode*

Inherited, Initial = *use-page-writing-mode*

`start-side-separation (DSSSL:-none-, CSS:-none-)` = length-specifier

Value(s): {*0..* available-size}

Inherited, Initial = *0.0pt*

`start-side-size (DSSSL:-none-, CSS:-none-)` = length-specifier

Value(s): {*0..page-height*}

Inherited, Initial = *0.0pt*

`start-side-overflow (DSSSL:-none-, CSS:overflow)` = (*visible* | *hidden* | *scroll* | *auto*)

Inherited, Initial = *auto*

`start-side-writing-mode (DSSSL:-none-, CSS:-none)` = a writing-mode-specifier | *use-page-writing-mode*

Inherited, Initial = *use-page-writing-mode*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**fo:simple-page-master**>

3.6.3 Formatting Object's Formal Specification

A **simple-page-master** is formatted to produce a sequence of page areas.

NOTE: The **simple-page-master** is intended for systems that wish to provide a very simple page layout facility. Future versions of this specification will support more complex page layouts constructed using the **page-master** and **column-set** formatting objects.

A **simple-page-master** shall be allowed only within the **page-sequence**.

Ed. Note: This restriction applies only to the July 1998 Draft.

The **`simple-page-master`** supports only the sequential-tiled-page-model, with an ordered set of up to 5 of the following areas: header, body, footer, end-side, and start-side. The user may specify the size (height or width) of the header, footer, end-side, and start-side areas and the separation distances between the adjacent areas. The height of the body area is the page's size (`page-height` for horizontal `writing-mode`s, and `page-width` for vertical `writing-mode`s) minus the sum of the header & footer heights, the separations between the areas, and the page's margin in the block-progression-direction.

The stacking direction of the areas, the page and area heights and separation distances are in the direction specified by the `writing-mode`'s block-progression-direction.

The width of each area is the full available distance in the inline-progression-direction after subtracting the page's margin (and may not be negative).

A **`simple-page-master`** may use up to 6 associated **`queue`**s. These queues are not direct children of the page-sequence (but are associated with it by name or via an explicit mapping table).

title

A **`queue`** with `queue-name=`*`title`*.

For online presentations only, this object holds a single title textline to be presented in the window title bar when this **`simple-page-master`** is being viewed.

If provided for print environments, this object is ignored.

If there is too much text for the title area, the browser may truncate the presentation.

The content of a title is repeated on each page by replaying the title queue after the body area is processed. (This allows for proper presentation of "dictionary"-style running headers/footers.)

header

A **`queue`** with `queue-name=`*`header`*.

Holds the content to be placed in the header area(s).

For print and online environments, this object holds a set of information that is presented in a separate area at the top of the page or window.

> **Ed. Note:** As defined by writing-mode, we need a term for relative directions that is invariant across inline and block-level objects.

If there is too much text for the header area, the presentation may be truncated/clipped.

The content of a header is repeated on each page by replaying the header queue after the body area is processed. (This allows for proper presentation of "dictionary"-style running headers/footers.)

footer

A **queue** with `queue-name=`*`footer`*.

Holds the content to be placed in the footer area(s).

For print and online environments, this holds a set of information that is presented in a separate area at the bottom of the page or window.

If there is too much text for the footer area, the presentation may be truncated/clipped.

The content of a footer is repeated on each page by replaying the footer queue after the body area is processed. (This allows for proper presentation of "dictionary"-style running headers/footers.)

start-side

A **queue** with `queue-name=`*`start-side`*.

Holds the content to be placed in the start-side area(s).

For print and online environments, this object holds a set of information that is presented in a separate area at the starting edge (as specified by the `page-writing-mode` property) of the page or window.

If there is too much text for the start-side area, the presentation may be truncated/clipped.

The content of a start-side area is repeated on each page by replaying the start-side queue after the body area is processed. (This allows for proper presentation of "dictionary"-style running headers/footers.)

end-side

A **queue** with `queue-name=`*`end-side`*.

Holds the content to be placed in the end-side area(s).

For print and online environments, this holds a set of information that is presented in a separate area at the ending edge of the page or window.

If there is too much text for the end-side area, the presentation may be truncated/clipped.

The content of a end-side area is repeated on each page by replaying the end-side queue after the body area is processed. (This allows for proper presentation of "dictionary"-style running headers/footers.)

body

A **queue** with `queue-name=`*`body`*.

Holds the content to be placed in the body area(s).

For print and online environments, this holds the information that is presented in the main area in the middle of the page or window.

In a print environment, if there is too much text for the body area the formatter should create additional pages until all the content is presented.

In a online environment, if there is too much text for the body area the formatter can create additional pages/frames/panels until all the content is presented or it can present the content in a scrolling view.

3.6.4 To Resolve

Should defaults be in points? CSS has made the recommendation that pixels are preferred due to issues with browsers & video-drivers. The print industry wants a "real" measurements system, such as points or mm. Points are not nationally biased and the "computer point" has been widely accepted.

How to handle left, right, and centered header & footer areas in the simple-page-master.

3.7 Queue Flow Object

NOTE: Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.7.1 Purpose

A **`queue`** is used to gather content flow objects to be assigned to (placed into) a given area or set of chained-areas.

3.7.2 Formatting Object Summary

<**`fo:queue`**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`queue-name (DSSSL:-none-, CSS:-none-)` = name-specifier

Required

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**`fo:queue`**>

3.7.3 Formatting Object's Formal Specification

A **`queue`** shall not be allowed within the content of any formatting object except a **`page-sequence`**.

> **Ed. Note:** This restriction applies only to the July 1998 Draft.

The **`queue`** holds a sequence or tree of formatting-objects that is to be presented in a like-named area of the layout defined by the **`simple-page-master`**.

The following properties apply to a **`queue`**:

- `queue-name` specifies the area name in the parent object into which this object's content will be placed.

  For the **`simple-page-master`**, the queue-name may be: *`title`*, *`header`*, *`body`*, *`footer`*, *`start-side`*, or *`end-side`*. If two or more **`queue`**s have the same queue-name, the like-named queues will be merged into a single queue.

3.8 Sequence Flow Object

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.8.1 Purpose

A **`sequence`** is used to group flow objects and to assign inherited properties to be shared across them. (Note the difference between a **`queue`** and a **`sequence`**.)

3.8.2 Formatting Object Summary

<**`fo:sequence`**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**`fo:sequence`**>

3.8.3 Formatting Object's Formal Specification

A **`sequence`** formatting object is formatted to produce the series of the areas produced by each of its children. This object holds its content as children. Its children may be all inline, all block-level, or a mixture of inline & block-level, if so allowed within the sequence's parent.

> **NOTE:** A **`sequence`** is useful for specifying inherited properties. For example, a **`sequence`** with a specification of a `font-style` property may be constructed for an italic-emphasis phrase element in a **`block`**.

A **`sequence`** shall accept a formatting object if and only if its parent would accept the formatting objects in that **`sequence`**.

A **`sequence`** has no applicable properties.

3.9 Block Flow Object

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.9.1 Purpose

A **`block`** formatting object allows the formatter to create a block-level area that contains textlines.

This object is commonly used for formatting paragraphs, titles, headlines, figure and table captions, etc.

It normally specifies a rectangular area that occupies the width of the containing area and a height that is determined by the amount of text that the **`block`** contains.

A **`block`** may specify separation between it and a preceding block-level object or subsequent block-level object as well as unique indents on the start of the first textline of the **`block`** and end of the last textline of the **`block`**.

3.9.2 Formatting Object Summary

`<fo:block`

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`language (DSSSL:language & D:country, CSS:-none-)` = (*`none`* | *`use-document`* | *`an xml:lang specifier`*)

Inherited, Initial = *`use-document`*

NOTE: The value *`use-document`* specifies one should use the language/country/script specified in the source document’s xml:lang specifier. -- An explicit value has the same form as the xml:lang specifier and overrides the language derived through xml:lang. -- Choosing "none" disables hyphenation and forces a simple line-breaking strategy. Used for program text and poetry.

`background-attachment` = (*scroll* | *fixed*)

Inherited, Initial = *scroll*

`background-color` = a color-specifier or *transparent*

Inherited, Initial = *transparent*

`background-image` = a URI or *none*

Inherited, Initial = *none*

`background-position-x` = (*a length-specifier* | *left* | *center* | *right*)

Value(s): {*0..max-length*}

Inherited, Initial = *left*

`background-position-y` = (*a length-specifier* | *top* | *middle* | *bottom*)

Value(s): {*0..max-length*}

Inherited, Initial = *top*

`background-repeat` = (*no-repeat* | *repeat* | *repeat-x* | *repeat-y*)

Inherited, Initial = *repeat*

`font-family (DSSSL:font-family-name, CSS:-same-)` = font-name or font-name-list

Inherited, Initial = *any*

`font-style (DSSSL:font-posture, CSS:-same-)` = (*normal* | *italic* | *oblique* | *-TBD- (check CSS)*)

Inherited, Initial = *normal*

`font-stretch (DSSSL:font-proportionate-width, CSS:-same-)` = (*ultra-condensed* | *extra-condensed* | *condensed* | *semi-condensed* | *normal* | *semi-expanded* | *expanded* | *extra-expanded* | *ultra-expanded*)

Inherited, Initial = *normal*

`font-size (DSSSL:-same-, CSS:-same-)` = length-specifier

Value(s): {*1.0pt...1024.0pt*}

Inherited, Initial = *10.0pt (DSSSL)*

`font-size-adjust (DSSSL:-none-, CSS:-same-)` = length-specifier

Value(s): {*TBD*...*TBD*}

Inherited, Initial = *TBD*

`font-variant (DSSSL:-none-, CSS:-same-)` = (*normal* | *small-caps*)

Inherited, Initial = *normal*

`font-weight (DSSSL:-same-, CSS:-same-)` = (*any* | *not-applicable* | *ultra-light* | *extra-light* | *light* | *semi-light* | *book (added)* | *normal (added)* | *medium* | *semi-bold* | *bold* | *extra-bold* | *ultra-bold* | *... (See CSS & PANOSE)*)

Inherited, Initial = *normal*

`glyph-alignment-mode (DSSSL:-same-)` = (*base* | *center* | *top* | *bottom* | *font*)

Inherited, Initial = *font*

NOTE: Used to set the textline's placement-path position relative to the origin of the block-level area. (See the extended description of "Textline Spacing", following this section.) Used to set alignment-line or placement-path. CSS supports only *font*.
Non-core

`indent-end (DSSSL:end-indent, CSS:-object-margin-)` = length-specifier

Value(s): {*0.0pt*...available-width}

Inherited, Initial = *0.0pt*

`indent-start (DSSSL:start-indent, CSS:-object-margin-)` = length-specifier

Value(s): {*0.0pt*...available-width}

Inherited, Initial = *0.0pt*

`indent-first-line-start (DSSSL:first-line-start-indent, CSS:text-indent)` = length-specifier

Value(s): {-indent-start...available-width}

Optional (Non-inherited), Default = *0.0pt*

NOTE: (CSS/DSSSL differ) Confirm that this is ADDED to the indent-start, not a replacement.

`break-after (DSSSL:-same-)` = (*none* | *page* | *page-odd* | *page-even*)

Optional (Non-inherited), Default = *none*

`break-before (DSSSL:-same-)` = (*none* | *page* | *page-odd* | *page-even*)

Optional (Non-inherited), Default = *none*

`keep (DSSSL:-same-)` = (*auto* | *no-break* | *page*)

Optional (Non-inherited), Default = *auto*

NOTE: A value of *no-break* specifies the object may not be broken. A value of *auto* specifies that it may be broken in accordance with the widow/orphan specifier. All other values indicate that this object shall be together as a single unit if it would break over the boundary indicated by the property value.

`orphans (DSSSL:orphan-count, CSS:orphans)` = integer

Value(s): {*0...999*}

Inherited, Initial = *2*

`widows (DSSSL:widow-count, CSS:widows)` = integer

Value(s): {*0...999*}

Inherited, Initial = *2*

`keep-with-next (DSSSL:-same-)` = (*true* | *false*)

Optional (Non-inherited), Default = *false*

`keep-with-previous (DSSSL:-same-)` = (*true* | *false*)

Optional (Non-inherited), Default = *false*

`block-line-breaking (DSSSL:lines)` = (*wrap* | *asis* | *as-is-wrap* | *asis-truncate* | *none*)

Inherited, Initial = *wrap*

`block-asis-truncate-indicator (DSSSL:asis-truncate-char)` = (*none* | *a character*)

Inherited, Initial = *none*

NOTE: `Non-core`

`block-asis-wrap-indicator (DSSSL:asis-wrap-char)` = (*none* | *a character*)

Inherited, Initial = *none*

NOTE: `Non-core`

`block-asis-wrap-indent (DSSSL:asis-wrap-indent)` = length-specifier

Value(s): {*-indent-start*...*-TBD-*}

Inherited, Initial = *0.0pt*

NOTE: `Non-core`

`hyphenation-keep (DSSSL:-same-)` = (*none* | *spread* | *page* | *column*)

Inherited, Initial = *none*

NOTE: `Non-core`

`hyphenation-ladder-count (DSSSL:-same-)` = integer

Value(s): {*1*...*999*}

Inherited, Initial = *2*

NOTE: `Non-core`

`text-align (DSSSL:quadding, CSS:text-align)` = (*start* | *end* | *left* | *right* | *spread-inside* | *spread-outside* | *page-inside (Defer)* | *page-outside (Defer)* | *center* | *justify* | *justify-force*)

Inherited, Initial = *start*

`text-align-last (DSSSL:last-line-quadding, CSS:-none-)` = (*auto* | *start* | *end* | *left* | *right* | *spread-inside* | *spread-outside* | *page-inside (Defer)* | *page-outside (Defer)* | *center* | *justify*)

Inherited, Initial = *start*

NOTE: `Non-core`

`linespacing-strategy (DSSSL: #f on min-leading)` = (*fixed* | *auto*)

Inherited, Initial = *auto*

`linespacing (DSSSL:line-spacing)` = length-specifier

Value(s): {*0.0pt*...*-TBD-*}

Inherited, Initial = *12.0pt*

`space-after-maximum (DSSSL:space-after)` = length-specifier

Value(s): {*--TBD-...-TBD-*}

Optional (Non-inherited), Default = *0.0pt*

NOTE: `Non-core`

`space-after-minimum (DSSSL:space-after)` = length-specifier

Value(s): {*--TBD-...-TBD-*}

Optional (Non-inherited), Default = *0.0pt*

NOTE: `Non-core`

`space-after-optimum (DSSSL:space-after)` = length-specifier

Value(s): {*--TBD-...-TBD-*}

Optional (Non-inherited), Default = *0.0pt*

`space-before-maximum (DSSSL:space-before)` = length-specifier

Value(s): {*--TBD-...-TBD-*}

Optional (Non-inherited), Default = *0.0pt*

NOTE: `Non-core`

`space-before-minimum (DSSSL:space-before)` = length-specifier

Value(s): {*--TBD-...-TBD-*}

Optional (Non-inherited), Default = *0.0pt*

NOTE: `Non-core`

`space-before-optimum (DSSSL:space-before)` = length-specifier

Value(s): {*--TBD-...-TBD-*}

Optional (Non-inherited), Default = *0.0pt*

`writing-mode (DSSSL:-same-)` = writing-mode-specifier

Inherited, Initial = *lr-tb*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**fo:block**>

3.9.3 Formatting Object's Formal Specification

A **block** is a *block-level* formatting object.

A **block** directly contains its children, which may be a mixture of *inline* or *block-level* formatting objects.

- *Inline* child formatting objects within a **block** are formatted to produce one or more textline areas. Multiple *inline* objects may be placed successively into a single textline. *Inline* objects may (or may not) be split across two or more textlines if necessary (and if allowed to split) if the *inline* does not fit in the remaining space in the textline.

- *Block-level* child formatting objects within a **block** implicitly specify line-breaks before and after the *block-level* object. Each child *block-level* produces a single area which is treated by the formatter of the **block** as if it were a textline area. These areas shall be added to the resulting sequence of areas within the **block**.

  NOTE: This specifies that users may nest a **block** inside another **block**. When this happens the outer **block** does not end before the nested **block**, it is simply suspended. The normal mid-block quadding and indents apply to the last textline prior to the nested **block**'s area. Similarly, the outer **block** resumes after the nested **block** without a new first-textline indent.

  NOTE: Typically, a break implies that a new textline is to be started.

  The shift-direction for *inline* areas in the **block** is the reverse of the line-progression-direction of the **block**.

3.9.4 To Resolve

The following properties apply to the **block**:

- `country (Combine with language)`, `language (DSSSL:language & D:country)`, & `script (DSSSL:-none-)`

  Replaced with xml:lang. Used for justification and hyphenation (Can extract from XML if the XML identifier is comprised of 2 or more tokens, each of which are 2 chars RFC-1766 (1995)).

- Need to evaluate RFC-1766 issues

3.10 Character Flow Object

NOTE: Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.10.1 Purpose

The **`character`** formatting object is used when one needs to explicitly override a specific character or array of characters with a specific glyph.

When the result tree is interpreted as a tree of formatting objects, a character in the result tree is treated as if it were an empty element of type **`fo:character`** with a `char` attribute equal to the character.

3.10.2 Formatting Object Summary

<**`fo:character`**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`background-attachment` = (*`scroll`* | *`fixed`*)

Inherited, Initial = *`scroll`*

`background-color` = a color-specifier or *`transparent`*

Inherited, Initial = *`transparent`*

`background-image` = a URI or *`none`*

Inherited, Initial = *`none`*

`background-position-x` = (*`a length-specifier`* | *`left`* | *`center`* | *`right`*)

Value(s): { *`0..max-length`* }

Inherited, Initial = *`left`*

`background-position-y` = (*`a length-specifier`* | *`top`* | *`middle`* | *`bottom`*)

Value(s): { *`0..max-length`* }

Inherited, Initial = *`top`*

`background-repeat` = (*`no-repeat`* | *`repeat`* | *`repeat-x`* | *`repeat-y`*)

Inherited, Initial = *`repeat`*

`text-shadow (DSSSL:-none-, CSS:-same-)` = see CSS

Value(s): { *`-TBD-...-TBD-`* }

Inherited, Initial = *-TBD-*

`text-transform (DSSSL:-none-, CSS:-same-)` = (*as-entered* | *lower* | *upper* | *title* | *(see CSS)*)

Inherited, Initial = *as-entered*

/>

3.10.3 Formatting Object's Formal Specification

A **`character`** formatting object is atomic.

A **`character`** can only be *`inline`*.

> **NOTE:** A character string can be implemented as a **`sequence`** of **`character`**s.

A **`character`** is formatted to produce a single *`inline`* area. This may be merged with adjacent *`inline`* areas if the `ligature` property is *`true`*. The position-point of the *`inline`* area is the `position-point` of the glyph specified in the font resource for the specified `writing-mode`. The escapement direction is the direction between the position-point and escapement-points as specified in the font resource for the specified `writing-mode`. The size of the area in the inline-progression-direction is the distance between the position and escapement-points. [[Ed-Note: How does this deal with backgrounds when test is letterspaced or kerned?]] The size of the area before and after the placement-path in the shift-direction is the smallest that will enclose the extent of the glyph in those directions as specified in the font resource for the specified `writing-mode`. If the nominal alignment mode of the font resource for the **`character`**'s `writing-mode` is not the same as the **`block`**'s alignment mode, then the glyph area is automatically adjusted as specified by the `alignment-mode` in the font resource for the specified `writing-mode`.

3.10.4 To Resolve

The following properties apply to a **`character`**:

- `char`
- `char-kern`
- `char-kern-mode`
- `char-ligature`
- `color (DSSSL:-same-)`
- font-specification (to be supplied in future Drafts)
- `glyph-alignment-mode (DSSSL:-same-, CSS:-none-)` **Non-core**
- `hyphenate`

- `hyphenation-char (DSSSL:-same-, CSS:-none-)`
- `inhibit-wrap`
- `language (DSSSL:language & D:country)`
- `position-point-shift`
- `letterspace-after-maximum (DSSSL:inline-space-after, CSS:letter-spacing)`
- `letterspace-after-minimum (DSSSL:inline-space-after, CSS:letter-spacing)`
- `letterspace-after-optimum (DSSSL:inline-space-after, CSS:letter-spacing)`
- `wordspacing-maximum (DSSSL:inline-space-space, CSS:word-spacing)`
- `wordspacing-minimum (DSSSL:inline-space-space, CSS:word-spacing)`
- `wordspacing-optimum (DSSSL:inline-space-space, CSS:word-spacing)`
- `text-shadow (DSSSL:-none-, CSS:-same-)`
- `text-transform (DSSSL:-none-, CSS:-same-)` Add from CSS: *`capitalize`* | *`uppercase`* | *`lowercase`* | *`none`*
- `writing-mode (DSSSL:-same-)`

3.11 List Flow Object

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

> **Ed. Note:** Lists are not completely specified in this draft. The formatting of lists involves the positioning and formatting of the list-item-label before and (usually) adjacent to the first line of text in the list-item-body. CSS and DSSSL use different models to describe this formatting behavior. They also support differing formatting when the label becomes too wide to fit in the allowed space. We plan to resolve these issues in a future draft.

3.11.1 Purpose

A **`list`** is used to group all the items in a list.

A **`list`** may be nested within another list, and it is a block-level object.

`List`s are useful for controlling the separation between preceding and following formatting objects through the `space-before-minimum` and `space-after-minimum` properties . They can be also used to specify the `indent` and `margins` for nested lists, and for controlling the break preferences. They also provide a mechanism for specifying and bracketing autonumbering sequences.

3.11.2 Formatting Object Summary

<fo:list

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *none*

`background-attachment` = (*scroll* | *fixed*)

Inherited, Initial = *scroll*

`background-color` = a color-specifier or *transparent*

Inherited, Initial = *transparent*

`background-image` = a URI or *none*

Inherited, Initial = *none*

`background-position-x` = (*a length-specifier* | *left* | *center* | *right*)

Value(s): {*0..max-length*}

Inherited, Initial = *left*

`background-position-y` = (*a length-specifier* | *top* | *middle* | *bottom*)

Value(s): {*0..max-length*}

Inherited, Initial = *top*

`background-repeat` = (*no-repeat* | *repeat* | *repeat-x* | *repeat-y*)

Inherited, Initial = *repeat*

`break-before` = (*none* | *page* | *page-odd* | *page-even* | Optional (Non-inherited), Default = *none*

NOTE: Complex mapping to CSS. Specifies whether a page, column, etc., should start before the **`list`**.

`break-after` = (*none* | *page* | *page-odd* | *page-even* | Optional (Non-inherited), Default = *none*

NOTE: Complex mapping to CSS. Specifies whether a page, column, etc., should start after the **`list`**.

`indent-start` = length-specifier

Value(s): {*0.0pt*...available-width}

Inherited, Initial = *0.0pt*

NOTE: Specifies the initial point of indentation for all members of the **`list`**.

`indent-end` = length-specifier

Value(s): {*0.0pt*...available-width}

Inherited, Initial = *0.0pt*

NOTE: Specifies the final point of indentation for all members of the **`list`**. In a left to right writing direction languages, this is normally called right-indent.

`space-before-maximum` = length-specifier

Value(s): {*0.0pt*...available-length}

Inherited, Initial = *0.0pt*

NOTE: Specifies the amount of space to be inserted before the **`list`** in the block-progression-direction. **`Non-core`**

`space-before-minimum` = length-specifier

Value(s): {*0.0pt*...available-length}

Inherited, Initial = *0.0pt*

NOTE: Specifies the amount of space to be inserted before the **`list`** in the block-progression-direction. **`Non-core`**

`space-before-optimum` = length-specifier

Value(s): {*0.0pt*...available-length}

Inherited, Initial = *0.0pt*

NOTE: Specifies the amount of space to be inserted before the **`list`** in the block-progression-direction.

`space-after-maximum` = length-specifier

Value(s): {*0.0pt*...available-length}

Inherited, Initial = *0.0pt*

NOTE: Specifies the amount of space to be inserted after the **`list`** in the block-progression-direction. **`Non-core`**

`space-after-minimum` = length-specifier

Value(s): {*`0.0pt`*...available-length}

Inherited, Initial = *`0.0pt`*

NOTE: Specifies the amount of space to be inserted after the **`list`** in the block-progression-direction. **`Non-core`**

`space-after-optimum` = length-specifier

Value(s): {*`0.0pt`*...available-length}

Inherited, Initial = *`0.0pt`*

NOTE: Specifies the amount of space to be inserted after the **`list`** in the block-progression-direction.

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**`fo:list`**>

3.11.3 Formatting Object's Formal Specification

A **`list`** is a block-level flow object that contains one or more **`list-item`** objects.

Although a list can be nested inside another list, it cannot be a direct child; rather, it can be the child of a list's **`list-item-body`**.

3.12 List-Item Flow Object

3.12.1 Purpose

A **`list-item`** flow object contains the label and the body of each item; it may be used for overriding and modifying some of the list's properties on a case by case basis.

3.12.2 Formatting Object Summary

<**`fo:list-item`**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`background-attachment` = (*`scroll`* | *`fixed`*)

Inherited, Initial = *scroll*

`background-color` = a color-specifier or *transparent*

Inherited, Initial = *transparent*

`background-image` = a URI or *none*

Inherited, Initial = *none*

`background-position-x` = (*a length-specifier* | *left* | *center* | *right*)

Value(s): {*0..max-length*}

Inherited, Initial = *left*

`background-position-y` = (*a length-specifier* | *top* | *middle* | *bottom*)

Value(s): {*0..max-length*}

Inherited, Initial = *top*

`background-repeat` = (*no-repeat* | *repeat* | *repeat-x* | *repeat-y*)

Inherited, Initial = *repeat*

`indent-start` = length-specifier

Value(s): {*0.0pt*...available-width}

Inherited, Initial = *0.0pt*

NOTE: Specifies the initial point of indentation for the whole **`list-item`** Overrides the value set in the **`list`**.

`indent-end` = length-specifier

Value(s): {*0.0pt*...available-width}

Inherited, Initial = *0.0pt*

NOTE: Specifies the final point of indentation for the whole **`list-item`** Overrides the value set for the **`list`**.

`item-space-before-maximum` = length-specifier

Value(s): {*0.0pt*...available-length}

Inherited, Initial = *0.0pt*

NOTE: Specifies the amount of space to be inserted before each **`list item`** in the direction of the block-progression-direction. Usually specified at the list level. Note: This is a block-level space. **`Non-core`**

`item-space-before-minimum` = length-specifier

Value(s): {*`0.0pt`*...available-length}

Inherited, Initial = *`0.0pt`*

NOTE: Specifies the amount of space to be inserted before each **`list item`** in the direction of the block-progression-direction. Usually specified at the list level. Note: This is a block-level space.**`Non-core`**

`item-space-before-optimum` = length-specifier

Value(s): {*`0.0pt`*...available-length}

Inherited, Initial = *`0.0pt`*

NOTE: Specifies the amount of space to be inserted before each **`list item`** in the direction of the block-progression-direction. Usually specified at the list level. Note: This is a block-level space.

`item-space-after-maximum` = length-specifier

Value(s): {*`0.0pt`*...available-length}

Inherited, Initial = *`0.0pt`*

NOTE: Specifies the amount of space to be inserted after each **`list item`** in the direction of the block-progression-direction. Usually specified at the list level. Note: This is a block-level space. **`Non-core`**

`item-space-after-minimum` = length-specifier

Value(s): {*`0.0pt`*...available-length}

Inherited, Initial = *`0.0pt`*

NOTE: Specifies the amount of space to be inserted after each **`list item`** in the direction of the block-progression-direction. Usually specified at the list level. Note: This is a block-level space. **`Non-core`**

`item-space-after-optimum` = length-specifier

Value(s): {*`0.0pt`*...available-length}

Inherited, Initial = *0.0pt*

NOTE: Specifies the amount of space to be inserted after each **`list item`** in the direction of the block-progression-direction. Usually specified at the list level. Note: This is a block-level space.

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**`fo:list-item`**>

3.12.3 Formatting Object's Formal Specification

A list-item flow object can only be contained by a list. It is a wrapper for a **`list-item-label`** and an **`list-item-body`**. It controls their position relative to other items within the list. Most of its properties are typically specified on the **`list`**. It controls the position and padding of the label and the body within the list-item and in relation to other list-items in the list.

3.13 List-Item-Label Flow Object

NOTE: Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.13.1 Purpose

A **`list-item-label`** is used to either enumerate, identify or adorn the **`list-item`**'s body.

3.13.2 Formatting Object Summary

<**`fo:list-item-label`**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *none*

`background-attachment` = (*scroll* | *fixed*)

Inherited, Initial = *scroll*

`background-color` = a color-specifier or *transparent*

Inherited, Initial = *transparent*

`background-image` = a URI or *none*

Inherited, Initial = *none*

`background-position-x` = (*a length-specifier* | *left* | *center* | *right*)

Value(s): {*0*..*max-length*}

Inherited, Initial = *left*

`background-position-y` = (*a length-specifier* | *top* | *middle* | *bottom*)

Value(s): {*0*..*max-length*}

Inherited, Initial = *top*

`background-repeat` = (*no-repeat* | *repeat* | *repeat-x* | *repeat-y*)

Inherited, Initial = *repeat*

`label-width` = length-specifier

Value(s): {*0.0pt*...available-length}

Inherited, Initial = *0.0pt*

NOTE: Specifies the amount of space, in the direction of the writing mode, to be reserved for the label. Typically set at the list level. [[Ed-Note: Check the issue of value=0]] If the value is 0, the formatter should calculate the width on the basis of the longest label in the **`list`**.

`space-end` = length-specifier

Value(s): {*0.0pt*...available-width}

Inherited, Initial = *0.0pt*

NOTE: Note: Specifies the minimum space in the direction of the block-progression-direction between the label and the body's first line.

`label-separator` = length-specifier

Value(s): {-TBD-..-TBD-}

Inherited, Initial = *12.0pt*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**`fo:list-item-label`**>

3.13.3 Formatting Object's Formal Specification

A **`list-item-label`** can be contained only in a **`list-item`**. It can be used for enumerating the **`list-item`**. It can control the positioning of the label and its placement with respect tot he **`list-item-body`**. The label has content, and is formatted to become the adornment or enumeration of the **`list-item`**.

3.14 List-Item-Body Flow Object

3.14.1 Purpose

The `item-body` flow object holds the components (usually blocks) for a list item.

It controls styling defaults for the body, the spacing between lines and between paras within the list item, break precedences for line and paragraphs within the list item.

3.14.2 Formatting Object Summary

<**`fo:list-item-body`**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`background-attachment` = (*`scroll`* | *`fixed`*)

Inherited, Initial = *`scroll`*

`background-color` = a color-specifier or *`transparent`*

Inherited, Initial = *`transparent`*

`background-image` = a URI or *`none`*

Inherited, Initial = *`none`*

`background-position-x` = (*`a length-specifier`* | *`left`* | *`center`* | *`right`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`left`*

`background-position-y` = (*`a length-specifier`* | *`top`* | *`middle`* | *`bottom`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`top`*

`background-repeat` = (*`no-repeat`* | *`repeat`* | *`repeat-x`* | *`repeat-y`*)

Inherited, Initial = *`repeat`*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**fo:list-item-body**>

3.14.3 Formatting Object's Formal Specification

The item's body contains the content of the item, generally in the form of blocks.

3.15 Rule-Graphic Flow Object

NOTE: Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.15.1 Purpose

A **`rule-graphic`** is used to draw a graphic-line that is used to divide space on the page.

3.15.2 Formatting Object Summary

<**fo:rule-graphic**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *none*

`background-attachment` = (*scroll* | *fixed*)

Inherited, Initial = *scroll*

`background-color` = a color-specifier or *transparent*

Inherited, Initial = *transparent*

`background-image` = a URI or *none*

Inherited, Initial = *none*

`background-position-x` = (*a length-specifier* | *left* | *center* | *right*)

Value(s): {*0..max-length*}

Inherited, Initial = *left*

`background-position-y` = (*a length-specifier* | *top* | *middle* | *bottom*)

Value(s): {*0..max-length*}

Inherited, Initial = *top*

background-repeat = (*no-repeat* | *repeat* | *repeat-x* | *repeat-y*)

Inherited, Initial = *repeat*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**fo:rule-graphic**>

3.15.3 Formatting Object's Formal Specification

3.15.4 To Resolve

A **rule-graphic**s may be *inline* or *block-level*.

The following properties apply to the **rule-graphic**:

- color (DSSSL:-same-)
- block-level-alignment
- break-after (DSSSL:-same-)
- break-before (DSSSL:-same-)
- graphic-line-thickness (DSSSL:line-thickness)
- indent-end (DSSSL:end-indent, CSS:-object-margin-)
- indent-start (DSSSL:start-indent, CSS:-object-margin-)
- inhibit-wrap
- keep (DSSSL:-same-)
- keep-with-previous (DSSSL:-same-)
- keep-with-next (DSSSL:-same-)
- rule-graphic-length
- rule-graphic-orientation
- position-point-shift
- space-after-maximum (DSSSL:space-after) **Non-core**
- space-after-minimum (DSSSL:space-after) **Non-core**
- space-after-optimum (DSSSL:space-after)

- `space-before-maximum (DSSSL:space-before)` **Non-core**
- `space-before-minimum (DSSSL:space-before)` **Non-core**
- `space-before-optimum (DSSSL:space-before)`
- `writing-mode (DSSSL:-same-)`

3.16 Graphic Flow Object

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.16.1 Purpose

Holds an image or vector graphic.

Placement in XSL may be inline or block-level.

Content of the graphic may be instream or external (linked).

3.16.2 Formatting Object Summary

<fo:graphic

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *none*

`background-attachment` = (*scroll* | *fixed*)

Inherited, Initial = *scroll*

`background-color` = a color-specifier or *transparent*

Inherited, Initial = *transparent*

`background-image` = a URI or *none*

Inherited, Initial = *none*

`background-position-x` = (*a length-specifier* | *left* | *center* | *right*)

Value(s): {*0..max-length*}

Inherited, Initial = *left*

`background-position-y` = (*a length-specifier* | *top* | *middle* | *bottom*)

Value(s): {*0..max-length*}

Inherited, Initial = *top*

`background-repeat` = (*no-repeat* | *repeat* | *repeat-x* | *repeat-y*)

Inherited, Initial = *repeat*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**fo:graphic**>

3.16.3 Formatting Object's Formal Specification

3.16.4 To Resolve

The **graphic** formatting object is a formatting wrapper to hold graphic objects.

A **graphic** may be *inline* or *block-level*.

A **graphic**'s content may be instream or external.

A **graphic** is not atomic.

The following properties apply to a **graphic**:

- `inline`
- `block-level-alignment`
- `break-after (DSSSL:-same-)`
- `break-before (DSSSL:-same-)`
- `color (DSSSL:-same-)`
- `external-graphic-id`
- `graphic-max-height`
- `graphic-max-width`
- `graphic-scale`
- `indent-end (DSSSL:end-indent, CSS:-object-margin-)`
- `indent-start (DSSSL:start-indent, CSS:-object-margin-)`
- `inhibit-wrap`

- keep (DSSSL:-same-)
- keep-with-previous (DSSSL:-same-)
- keep-with-next (DSSSL:-same-)
- position-point-x
- position-point-y
- position-preference (DSSSL:-same-)
- space-after-maximum (DSSSL:space-after) **Non-core**
- space-after-minimum (DSSSL:space-after) **Non-core**
- space-after-optimum (DSSSL:space-after)
- space-before-maximum (DSSSL:space-before) **Non-core**
- space-before-minimum (DSSSL:space-before) **Non-core**
- space-before-optimum (DSSSL:space-before)
- writing-mode (DSSSL:-same-)

3.17 Score Flow Object

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.17.1 Purpose

Highlights text. Used to produce underlines, strike-through, overbars, etc.

3.17.2 Formatting Object Summary

<**fo:score**

id (DSSSL:-none-, CSS:-none-) = id-specifier

Optional (Non-inherited), Default = *none*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**fo:score**>

3.17.3 Formatting Object's Formal Specification

3.17.4 To Resolve

This object holds its content as children.

The content is scored (a highlight or text decoration using a graphic line under, over, or through the text).

A **`score`** can contain only *`inline`* formatting objects.

> **NOTE:** Nest to apply multiple scoring (such as: over and underbar, double or triple underscore, double strikethrough, strikethrough and underscore, etc.)
>
> The CSS text-decoration property sets scoring and "blink". "Conforming user agents are not required to support blink."
>
> **Ed. Note:** Do we want/need to support blink?

The following properties apply to **`score`**:

- `color (DSSSL:-same-)`
- `graphic-line-thickness (DSSSL:line-thickness)`
- `position-shift`
- `inhibit-wrap`
- `score-spaces`

3.18 Inline-box Flow Object

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.18.1 Purpose

Highlights text or graphics.

Used to produce borders and backgrounds.

Controls spacing surrounding the content.

3.18.2 Formatting Object Summary

`<fo:inline-box`

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`background-attachment` = (*`scroll`* | *`fixed`*)

Inherited, Initial = *`scroll`*

`background-color` = a color-specifier or *`transparent`*

Inherited, Initial = *`transparent`*

`background-image` = a URI or *`none`*

Inherited, Initial = *`none`*

`background-position-x` = (*`a length-specifier`* | *`left`* | *`center`* | *`right`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`left`*

`background-position-y` = (*`a length-specifier`* | *`top`* | *`middle`* | *`bottom`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`top`*

`background-repeat` = (*`no-repeat`* | *`repeat`* | *`repeat-x`* | *`repeat-y`*)

Inherited, Initial = *`repeat`*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**`fo:inline-box`**>

3.18.3 Formatting Object's Formal Specification

3.18.4 To Resolve

The following properties apply to the **`inline-box`**:

- `box-reserve-space`
- `box-open-end`
- `box-size-after`
- `box-size-before`

- `box-type`
- `break-after (DSSSL:-same-)`
- `break-before (DSSSL:-same-)`
- `color (DSSSL:-same-)`
- `graphic-line-thickness (DSSSL:line-thickness)`
- `indent-end (DSSSL:end-indent, CSS:-object-margin-)`
- `indent-start (DSSSL:start-indent, CSS:-object-margin-)`
- `inhibit-textline-breaks?`
- `keep (DSSSL:-same-)`
- `keep-with-previous? (DSSSL:-same-)`
- `keep-with-next? (DSSSL:-same-)`
- `space-after-maximum (DSSSL:space-after)` **Non-core**
- `space-after-minimum (DSSSL:space-after)` **Non-core**
- `space-after-optimum (DSSSL:space-after)`
- `space-before-maximum (DSSSL:space-before)` **Non-core**
- `space-before-minimum (DSSSL:space-before)` **Non-core**
- `space-before-optimum (DSSSL:space-before)`
- `writing-mode (DSSSL:-same-)`

3.19 Block-level-box Flow Object

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.19.1 Purpose

Highlights text or graphics.

Used to produce borders and backgrounds.

Controls spacing surrounding the content.

3.19.2 Formatting Object Summary

<**fo:block-level-box**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *none*

`background-attachment` = (*scroll* | *fixed*)

Inherited, Initial = *scroll*

`background-color` = a color-specifier or *transparent*

Inherited, Initial = *transparent*

`background-image` = a URI or *none*

Inherited, Initial = *none*

`background-position-x` = (*a length-specifier* | *left* | *center* | *right*)

Value(s): {*0..max-length*}

Inherited, Initial = *left*

`background-position-y` = (*a length-specifier* | *top* | *middle* | *bottom*)

Value(s): {*0..max-length*}

Inherited, Initial = *top*

`background-repeat` = (*no-repeat* | *repeat* | *repeat-x* | *repeat-y*)

Inherited, Initial = *repeat*

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**fo:block-level-box**>

3.19.3 Formatting Object's Formal Specification

3.19.4 To Resolve

The following properties apply to the **block-level-box**:

- `box-reserve-space`
- `box-size-after`
- `box-size-before`

- box-type
- break-after (DSSSL:-same-)
- break-before (DSSSL:-same-)
- color (DSSSL:-same-)
- graphic-line-thickness (DSSSL:line-thickness)
- indent-end (DSSSL:end-indent, CSS:-object-margin-)
- indent-start (DSSSL:start-indent, CSS:-object-margin-)
- inhibit-textline-breaks?
- keep (DSSSL:-same-)
- keep-with-previous? (DSSSL:-same-)
- keep-with-next? (DSSSL:-same-)
- space-after-maximum (DSSSL:space-after) **Non-core**
- space-after-minimum (DSSSL:space-after) **Non-core**
- space-after-optimum (DSSSL:space-after)
- space-before-maximum (DSSSL:space-before) **Non-core**
- space-before-minimum (DSSSL:space-before) **Non-core**
- space-before-optimum (DSSSL:space-before)
- writing-mode (DSSSL:-same-)

3.20 Page-number Flow Object

> **NOTE:** Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.20.1 Purpose

This object is used to instruct the formatter to construct and present a page-number. (Page numbers can not be constructed by the XSL processor because it has no knowledge of the line-breaking or actual pagination, except in very limited cases.)

3.20.2 Formatting Object Summary

<fo:page-number

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`background-attachment` = (*`scroll`* | *`fixed`*)

Inherited, Initial = *`scroll`*

`background-color` = a color-specifier or *`transparent`*

Inherited, Initial = *`transparent`*

`background-image` = a URI or *`none`*

Inherited, Initial = *`none`*

`background-position-x` = (*`a length-specifier`* | *`left`* | *`center`* | *`right`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`left`*

`background-position-y` = (*`a length-specifier`* | *`top`* | *`middle`* | *`bottom`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`top`*

`background-repeat` = (*`no-repeat`* | *`repeat`* | *`repeat-x`* | *`repeat-y`*)

Inherited, Initial = *`repeat`*

/>

3.20.3 Formatting Object's Formal Specification

3.20.4 To Resolve

Multiple numbering sequences. (Examples: Front matter, body+appendix; Front-matter, by-chapter, by-appendix)

Complex numbers (15-5)

See **`xsl:number`** for properties.

3.21 Link Formatting Object

NOTE: Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.21.1 Purpose

A **`link`** formatting object creates an area that a user can select to request traversal to another resource. An XLink-aware processor may create these regions on its own, but that does not preclude a designer from creating additional links from contextual information in the document.

The **`link`** contains **`link-end-locator`** flow objects, which provide information about the destination or destinations of the **`link`**, as well as any renderable flow objects. The region of the **`link`** is coextensive with that of the renderable flow objects contained within it. A user agent may specify two modes of **`link`** selection: an informative one (such as hovering over the **`link`** with a mouse or tabbing to it) and an activating one (such as clicking on the **`link`** or pressing return). When the user selects the **`link`** for information, if there is only one link end, the content of that link end should be presented (in a tool tip, in the status bar, or any other means appropriate to the user agent); if there are multiple link ends, they may all be shown if the user agent has facility (such as a pop-up menu), or the user agent may simply indicate that multiple ends exist. When the user activates the **`link`**, a choice of link ends should be offered, using the renderable content of each link end. After the user makes a selection from that list, traversal can begin.

Since the **`link`** and its **`link-end-locator`**s are created by the XSL processor, link selection does not need to be automated at user action time. For instance, to randomly select one of a number of possible link ends, the stylesheet would simply create a single-ended link, whose **`link-end-locator`** is drawn randomly from the document, and present the user with only that choice.

XLink's transcluding capabilities can be handled simply by processing the target.

3.21.2 Formatting Object Summary

<**`fo:link`**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`background-attachment` = (*`scroll`* | *`fixed`*)

Inherited, Initial = *`scroll`*

`background-color` = a color-specifier or *`transparent`*

Inherited, Initial = *`transparent`*

`background-image` = a URI or *`none`*

Inherited, Initial = *`none`*

`background-position-x` = (*`a length-specifier`* | *`left`* | *`center`* | *`right`*)

Value(s): {*0..max-length*}

Inherited, Initial = *left*

`background-position-y` = (*a length-specifier* | *top* | *middle* | *bottom*)

Value(s): {*0..max-length*}

Inherited, Initial = *top*

`background-repeat` = (*no-repeat* | *repeat* | *repeat-x* | *repeat-y*)

Inherited, Initial = *repeat*

`merge-link-end-locators` = (*true* | *false*)

Optional (Non-inherited), Default = *true*

NOTE: Note: If this link formatting object occurs within another, and `merge-link-end-locators` is *true*, then the effect is the same as if the **`link-end-locator`**s of the ancestor were also **`link-end-locator`**s of this link. In other words, the **`link-end-locator`**s of the ancestor and those of this link are potential destinations when the user selects this link. If `merge-link-end-locators` is *false*, then only the **`link-end-locator`**s associated with this link are potential destinations from this link.

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**`fo:link`**>

3.21.3 Formatting Object's Formal Specification

The link formatting object simply defines the range of the selectable object. Any **`link-end-locator`** formatting objects contained within it, but not contained within any nested link formatting objects, are its applicable destinations. If the `merge-link-end-locators` characteristic is true, then the **`link-end-locator`**s of this particular link, as well as those of its ancestor link (and any merged therewith) are applicable destinations when the user selects this link. If `merge-link-end-locators` is false, then only the **`link-end-locator`**s specific to this link may be reached.

In a default stylesheet for XLink, an extended link group should create one link formatting object, while the locators should create **`link-end-locator`** formatting objects. A simple link should create a link formatting object and a **`link-end-locator`** formatting object. An XML IDREF element would also create a link and a **`link-end-locator`**.

3.22 Link-end-locator Formatting Object

NOTE: Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

3.22.1 Purpose

A **`link-end-locator`** formatting object identifies a resource which may be reached from its parent **`link`** formatting object. Its content is used to present the **`link-end-locator`** to the user, likely derived from the XLink title attribute.

See the description of **`link`** for more information.

3.22.2 Formatting Object Summary

<**`fo:link-end-locator`**

`id (DSSSL:-none-, CSS:-none-)` = id-specifier

Optional (Non-inherited), Default = *`none`*

`background-attachment` = (*`scroll`* | *`fixed`*)

Inherited, Initial = *`scroll`*

`background-color` = a color-specifier or *`transparent`*

Inherited, Initial = *`transparent`*

`background-image` = a URI or *`none`*

Inherited, Initial = *`none`*

`background-position-x` = (*`a length-specifier`* | *`left`* | *`center`* | *`right`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`left`*

`background-position-y` = (*`a length-specifier`* | *`top`* | *`middle`* | *`bottom`*)

Value(s): {*`0..max-length`*}

Inherited, Initial = *`top`*

`background-repeat` = (*`no-repeat`* | *`repeat`* | *`repeat-x`* | *`repeat-y`*)

Inherited, Initial = *`repeat`*

`href` = (*`XPointer`*)

Required

NOTE: The XPointer identifies the resource located by this **`link-end-locator`**. This may be taken directly from an attribute value (as for an XLink locator) or calculated (e.g., from an XML IDREF or from the content of a glossary reference).

`show-content` = (*`true`* | *`false`*)

Optional (Non-inherited), Default = *`false`*

NOTE: Note: Typically, the content of the **`link-end-locator`** formatting object is only shown so that the user may select a destination for the link. However, in some circumstances the designer may wish to present the title of the **`link-end-locator`** as part of the document's initial presentation.

NOTE: The user may also provide any additional inheritable properties for use by the descendants of this object.

> ... </**`fo:link-end-locator`**>

3.22.3 Formatting Object's Formal Specification

When a user requests traversal of a link formatting object, the user agent should present a list of possible destinations (if there is more than one). The user agent may also offer an informative mode of interaction with a link, such as hovering over it with a mouse pointer. In those cases, the formatted content of the **`link-end-locator`** formatting object should be used to represent this potential destination.

A default stylesheet for XLink would create a **`link-end-locator`** for every locator element; it would also create both a link and a **`link-end-locator`** for simple links. Similarly, an XML IDREF would generate both a link and a **`link-end-locator`**; an xref to a chapter might create a link whose only content is a **`link-end-locator`** to that chapter, whose content in turn is the formatted title of the chapter, and whose `show-content` characteristic is set to *`true`*.

3.23 Defined Terms

NOTE: Coordination between CSS and XSL properties and objects is an ongoing process, with the goal of defining a common underlying formatting model. Therefore, some of the object names and definitions, as well as property names, allowed values, and definitions may change as a result of this effort.

area

This is a DSSSL "area".

Ed. Note: Definition

boolean

A specifier having exactly 2 values:

`false`

Specifies the specifier is false.

`true`

Specifies the specifier is true.

NOTE: *`false`* & *`true`* may be used in other contexts. If so, their meaning will be described for that context.

color-specifier

Ed. Note: Use CSS Definition. Should we add CMYK and other models?

direction

The term direction should be qualified as follows:

column-progression-direction

Ed. Note: Defer -- Tables & Math

block-progression-direction

The direction of progression of sequential block-level object placements as specified by the writing-mode.

Perpendicular to the inline-progression-direction,

escapement-direction

The direction of progression of sequential glyph object placements along the placement-path as specified by the character/glyph information. May be overridden by the `escapement-direction` property. May be the same as or the reverse of the inline-progression-direction.

If unspecified, use the inline-progression-direction specified by the `writing-mode`.

inline-progression-direction

The direction of progression of sequential inline objects.

Usually the same direction as the escapement direction.

Perpendicular to the block-progression direction and the line-progression direction.

line-progression-direction

Perpendicular to the inline-progression-direction, the direction of successive textline placements as specified by the writing-mode.

Usually the same as the block-progression-direction.

row-progression-direction

Ed. Note: Defer -- Tables & Math

shift-direction

The direction of positive shift when characters, inline objects, or scores are shifted perpendicular to the placement-path. Usually the reverse of the line-progression-direction.

up-direction

Ed. Note: -TBD-.

starting-edge

Ed. Note: -TBD-.

ending-edge

Ed. Note: -TBD-.

ALSO

Ed. Note: Need orthogonal direction terms for ALL 4 edged of writing-mode-based directions. "Starting" is defined differently for inline and block-level formatting objects.

Ed. Note: Definition

id-specifier

is `none`, a unique-id, or a URI.

Specifies a unique identification of this object within the identifier-class or within the document, as specified in the individual property definition.

The following properties use id-specifiers:

- `id`
- `coalesce-id (DSSSL:-same- )[defer]`
- `external-graphic-id`
- `graphic-notation-id`
- `method-annotation-glyph [defer]`
- `method-emphasizing-mark [defer]`
- `method-glyph-reorder [defer]`
- `method-glyph-subst`
- `method-hyphenation (DSSSL:hyphenation-method) [defer]`
- `method-implicit-bidi (DSSSL:implicit-bidi-method) [defer]`
- `method-inline-note [defer]`
- `method-table-auto-width`
- `method-line-breaking (DSSSL:line-breaking-method) [defer]`
- `method-line-composition (DSSSL:line-composition-method) [defer]`

integer

This term may be qualified or unqualified, as follows:

integer

If unqualified, the term ALWAYS refers to an unsigned (positive) whole number. [0, 1, 2, ...]

The upper limit is system dependent, but shall be at least 32767.

integer, signed

An signed (positive or negative) whole number.

The lower limit is system dependent, but shall be at least -32767.

The upper limit is system dependent, but shall be at least 32767.

integer, strictly positive

An unsigned (positive) whole number greater than 0. [1,2,3...]

The upper limit is system dependent, but shall be at least 32767.

length-specifier

A distance specifier.

Must be qualified by a unit specification keyword, as follows:

cm

Metric centimeters

in

US inch (25.4 mm)

mm

Metric millimeters

pt

Computer printer's points, defined as exactly 1/72 of an inch.

NOTE: max-length is defined to be a value of at least 32767 points.

%

Percent (Individual properties that allow percentage values identify the measure that is used as the referent for this calculation)

px

Pixels (per CSS definition)

em

Fraction of current object's font-size. (See font-size property's description usage of EM and EX in setting font-size.)

ex

Fraction of current font's x-height.

max-length

is distance specifier allowing a value of at least 32767 points. If negative values are allowed this may have a negative value of at least 32767 points.

line

Since we have many kinds of lines, the term will be qualified as follows:

graphic-line

A graphic representation of a line-segment.

textline

a sequence of characters (and spaces) arranged along or relative to a common baseline.

queue

(This was "port" in the DSSSL specification.)

name-specifier

Specifies the name of an object, or more commonly is used to identify all members of a set of objects.

Name-specifiers may be a mixture of non-punctuation Unicode characters.

All like-named queue objects are merged into a single queue for the formatter.

Name-specifiers are used for:

- `auto-number-queue-name (DSSSL:-none-, CSS:-none-)`
- `queue-name (DSSSL:-none-, CSS:-none-)`
- `master-name (DSSSL:-none-, CSS:-none-)`

number

A signed numeric value that may include a fraction.

Ed. Note: Specification of "decimal radix character"?

page-model

Page designs can follow several models:

Sequential-tiled-page-model

This is your typical word processor page.

The subareas do not overlap. They are full width and are separated from the preceding subarea by a separation distance measured from the preceding area in the block-progression-direction specified by the writing-mode of the page.

Interlocking-tiled-page-model

This is your typical newspaper page.

The subareas do not overlap. Pages consist of rectangular, T, inverted-L shaped areas. They are non-overlapping and touch the adjacent areas (or page margins) on all sides.

Simple-freeform-page-model

The origins of the subareas are specified as X-Y coordinates measured from the page origin. Each area then specifies its shape relative to that origin. If areas overlap, they are overlapped in the order that the areas are specified (or in accordance with a `z-order` property), hiding the information of the underlying area.

Exclusionary-freeform-page-model

The origins of the subareas are specified as X-Y coordinates measured from the page origin. Each area then specifies its shape relative to that origin. If areas overlap, they are overlapped in the order that the areas are specified (or in accordance with a `z-order` property), reshaping the underlying area to wrap around the current area.

placement-path

A progression of items placed adjacently in the inline-progression-direction for inline objects or the block-progression-direction for block-level objects.

property

An attribute of a formatting object.

qualifier

An attribute of a character. Usually derived through system-dependent font metric and classification services.

string

An array of Unicode characters.

URI

Uniform Resource Identifier (an address of a resource [such as a file or component] on the WWW).

writing-mode-specifier

Specifies the orientation of the block-progression-direction, inline-progression-direction, line-progression-direction, shift-direction and writing-mode of the area/container. All orientations are absolute (based on the page orientation, not the containing area's orientation).

> **NOTE:** We are seeking input on writing directions. There is significant interest in making this standard international to the extent that it covers active languages. We have identified those writing directions that we believe to be in active use.

It is one of the symbols:

lr-tb

Specifies:

- an inline-progression-direction of left-to-right.
- a block-progression-direction and a line-progression-direction of top-to-bottom.
- a shift-direction of bottom-to-top.

rl-tb

Specifies:

- an inline-progression-direction of right-to-left.
- a block-progression-direction and a line-progression-direction of top-to-bottom.
- a shift-direction of bottom-to-top.

tb-rl

Specifies:

- an inline-progression-direction of top-to-bottom.
- a block-progression-direction and a line-progression-direction of right-to-left.
- a shift-direction of left-to-right.

lr-bt

Specifies:

- an inline-progression-direction of left-to-right.
- a block-progression-direction and a line-progression-direction of bottom-to-top.

- a shift-direction of bottom-to-top.

rl-bt

Specifies:

- an inline-progression-direction of right-to-left
- a block-progression-direction and a line-progression-direction of bottom-to-top.
- a shift-direction of bottom-to-top.

tb-lr

Specifies:

- an inline-progression-direction of top-to-bottom.
- a block-progression-direction and a line-progression-direction of left-to-right.
- a shift-direction of left-to-right.

bt-lr

Specifies:

- an inline-progression-direction of bottom-to-top.
- a block-progression-direction and a line-progression-direction of left-to-right.
- a shift-direction of -TBD-.

bt-rl

Specifies:

- an inline-progression-direction of bottom-to-top.
- a block-progression-direction and a line-progression-direction of right-to-left.
- a shift-direction of -TBD-.

lr-alternating-rl-tb

Specifies:

- an inline-progression-direction of left-to-right on odd lines and right-to-left on even lines.

- a block-progression-direction and a line-progression-direction of top-to-bottom.
- a shift-direction of -TBD-.

lr-alternating-rl-bt

Specifies:

- an inline-progression-direction of left-to-right on odd lines and right-to-left on even lines.
- a block-progression-direction and a line-progression-direction of bottom-to-top.
- a shift-direction of -TBD-.

lr-inverting-rl-tb

Specifies:

- an inline-progression-direction of left-to-right on odd lines with even lines completely inverted (r-l glyph order, inverted shift-direction and inverted up-vector).
- a block-progression-direction and a line-progression-direction of top-to-bottom.
- a shift-direction of -TBD-.

lr-inverting-rl-bt

Specifies:

- an inline-progression-direction of left-to-right on odd lines with even lines completely inverted (r-l glyph order, inverted shift-direction and inverted up-vector).
- a block-progression-direction and a line-progression-direction of bottom-to-top.
- a shift-direction of -TBD-.

tb-rl-in-rl-pairs

Specifies:

- an inline-progression-direction of -TBD-.
- a block-progression-direction and a line-progression-direction of -TBD-.
- a shift-direction of -TBD-.

Writing-mode-specifiers are used for the following properties:

- `writing-mode`
- `body-writing-mode`
- `end-side-writing-mode`
- `footer-writing-mode`
- `header-writing-mode`
- `page-writing-mode (DSSSL:-none-, CSS:-none-)`
- `start-side-writing-mode (DSSSL:-none-, CSS:-none-)`

3.24 Alphabetical Summary of Properties

> **NOTE:** We have just begun coordination between this formatting model and CSS, with the goal of defining a single underlying formatting model. Object names and definitions, as well as property names, allowed values, and definitions may change considerably as a result of that effort.

`background-attachment (DSSSL:-none-, CSS:-same-)`

From CSS: is one of the following:

`scroll`

The background-image will scroll with the contents of the viewport.

`fixed`

The background-image is presented at a fixed position. [Ednote: relative to what?, (origin in current context)]

The initial value is *`scroll`* (property is only active if the `background-image` is found).

[Ednote: Warning, scroll is meaningless for paged/print devices. Provide a better definition.]

`background-color (DSSSL:-same-differs, CSS:-same-)`

is one of the following:

`transparent`

Specifies that the *background-area* shall not be filled with a solid color, therefore any underlying information will show through.

a color-specifier

Specifies that the *background-area* shall be filled with the color specified.

(See also: `background-z-index` and `background-image`)

The default value is *`transparent`*.

`background-image (DSSSL:background-tile, CSS:-same-)`

is either *`none`* or a *URI*. Specifies an image that should be presented in the background.

The `background-attachment`, `background-repeat`, and `background-position` properties are used to control the formatting of a tiled background. The image is composed against the `background-color` or any underlying information.

The default value is *`none`*.

`background-position-x (DSSSL:-none-, CSS:background-position)`

is one of the following:

a percentage

[Ednote: Copy definition from CSS]

-or-

a length-specifier

[Ednote: Copy definition from CSS]

-or-

one of the following keywords

`left`

The left edge of the image is aligned with the left edge of the background area.

`center`

The center of the image is aligned with the center of the background area.

`right`

The right edge of the image is aligned with the right edge of the background area.

The default value is *`0%`* (left) (This property is only active if the `background-image` is found.)

`background-position-y (DSSSL:-none-, CSS:background-position)` is one of the following:

a percentage

[Ednote: Copy definition from CSS]

-or-

a length-specifier

[Ednote: Copy definition from CSS]

-or-

one of the following keywords

`top`

The top edge of the image is aligned with the top edge of the background area.

`middle`

The middle of the image is aligned with the middle of the background area.

`bottom`

The bottom edge of the image is aligned with the bottom edge of the background area.

The default value is *`0%`* (=top) (This property is only active if the `background-image` is found.)

`background-repeat (DSSSL:-none-, CSS:-same-)`

is one of the following:

`repeat`

The image specified by the `background-image` is repeated in both the x & y directions to fill the *background-area*.

`repeat-x`

The image specified by the `background-image` is repeated in the x direction to fill the *background-area*.

`repeat-y`

The image specified by the `background-image` is repeated in the y direction to fill the *background-area*.

no-repeat

The image specified by the `background-image` is not repeated.

The default value is *`repeat`*. (This property is only active if the `background-image` is found.)

`body-overflow (DSSSL:-none-, CSS:overflow)`

Specifies the overflow behavior for the body area.

(See `overflow`)

`body-writing-mode (DSSSL:-none-, CSS:-none-)`

a *writing-mode-specifier* or the value *`use-page-writing-mode`*.

Specifies the *writing-mode* within the body of a **`simple-page-master`**.

This *property* is inherited by the body-area's children as `writing-mode`.

The initial value is *`use-page-writing-mode`*.

`break-after (DSSSL:-same-, CSS:page-break-after)`

Specifies that a break after the *formatting object* is disallowed, allowed, or forced. On allowed and forced breaks, indicates that the *formatting object* shall start an *area* of the indicated type.

none

No break shall be allowed.

auto-page

Break shall be allowed but not forced. Should a break be necessary, this object shall be the last one placed in the current page. The object following this one shall begin a new page.

page

Break shall be forced, this object shall be the last one placed in the current page.

page-odd

Break shall be forced, this object shall be the last one placed in the current page, the next object shall be placed in a new odd-numbered page.

page-even

Break shall be forced, this object shall be the last one placed in the current page, the next object shall be placed in a new even-numbered page.

Should there be a conflict between `break-after` on the prior object and `break-before` on this object, the following precedence order applies. (The value closest to the top of the list shall be chosen.)

- *`page-odd`*
- *`page-even`*
- *`page`*
- *`auto-page`*
- *`none`*

This *property* is not inherited.

The default is *`none`*.

`break-before (DSSSL:-same-, CSS:page-break-before)`

Specifies that a break after the *formatting object* is disallowed, allowed, or forced. On allowed and forced breaks, indicates that the *formatting object* shall start an *area* of the indicated type.

`none`

No break shall be allowed.

`auto-page`

Break shall be allowed but not forced. Should a break be necessary, this object shall begin a new page.

`page`

Break shall be forced, this object shall begin a new page.

`page-odd`

Break shall be forced, this object shall be the first placed in a new odd-numbered page.

`page-even`

Break shall be forced, this object shall be the first placed in a new even-numbered page.

Should there be a conflict between `break-after` on the prior object and `break-before` on this object, the following precedence order applies. (The value closest to the top of the list shall be chosen.)

- *page-odd*
- *page-even*
- *page*
- *auto-page*
- *none*

This *property* is not inherited.

The default is *none*.

char

is a char-specifier. Specifies the Unicode character to be substituted/presented.

This *property* is not inherited.

1. If it is not specified, and there is a current node, and the current node has a char *qualifier*, then the value of the char *qualifier* shall be used as the value of this *property*.
2. If the value of the char-map *property* is not *none*, then it is applied to the value of the char *qualifier*, and the result is used as the value of the *property*.

This *property* may be used to control hyphenation as well as possibly being used in the selection of the glyph.

char-kern

is a *boolean*. Specifies whether kerning (*placement-adjustment*) is allowed.

If *true*, then kerning shall be performed according to the char-kern-mode *property*.

> **NOTE:** *Placement-adjustment* is not performed for glyphs whose placement-adjustment *qualifier* has the value *non-adjusting*.

The default value is *false*.

char-kern-mode

is one of the symbols

- *loose*,
- *normal*,
- *kern*,

- `tight`, or
- `touch`

specifying the *placement-adjustment* mode.

The default value is `normal`.

`char-ligature`

is a *boolean*. Specifies whether ligatures are allowed.

The default value is `false`.

`color (DSSSL:-same-)`

is a color-specifier that specifies the color in which the *formatting object*'s marks should be made.

This property may be inherited.

The initial value is the default color in the Device Gray color space.

[Ednote: Should color be split into 4 (or more) separate properties: "color-stroke", "color-fill", "color-background", and "color-shadow" (and ...)?]

`contents-alignment`

is one of the symbols `start`, `end`, `center`, or `justify`. Specifies the alignment of the child *area*s within the containing *area* in the *block-progression-direction* of the containing *area*.

The default value is `start`.

`contents-rotation`

is one of the integers 0, 90, 180, or 270. Specifies the counter-clockwise rotation to be applied to the *area* contents.

This *property* is not inherited.

The default is `0`.

`destination`

is either

`none`

[Ednote: -TBD-]

an address-specifier

or

a list of one or more objects of type address.

See .

This *property* is not inherited and shall be specified.

A value of *none* is used for a nested **link** and specifies that the contents of the *formatting object* shall not be considered part of the containing **link**.

`direction-embedded-text`

is one of the symbols *left-to-right* or *right-to-left*.

It shall be parallel to the `writing-mode` of the **block**.

This *property* is not inherited and shall be specified.

`inline`

is a *boolean*.

Specifies whether the *formatting object* is *inline* rather than *block-level*.

This *property* is not inherited.

The default value is *false*.

`end-side-overflow (DSSSL:-none-, CSS:overflow)`

Specifies the overflow behavior for the end-side area.

(See `overflow`)

`end-side-separation (DSSSL:-none-, CSS:-none-)`

is a *length-specifier*. Specifies the distance from the edge of the body area to the adjacent end-side area.

This *property* may be inherited.

The initial value is *18.0pt*.

`end-side-size (DSSSL:-none-, CSS:-none-)`

A length-specifier.

Specifies the width of the end-side area. If the corresponding **`queue`** content is absent, this space will still be reserved.

`end-side-writing-mode (DSSSL:-none-, CSS:-none-)`

a *writing-mode-specifier* or the value *`use-page-writing-mode`*.

Specifies the *writing-mode* within the end-side area of a **`simple-page-master`**.

This *property* is inherited by the end-side-area's children as `writing-mode`.

The initial value is *`use-page-writing-mode`*.

`external-graphic-id`

An *id-specifier*.

Specifies the entity containing the external graphic or *`none`* if the graphic is instream.

This *property* is not inherited and shall be specified.

Font-related properties

All font- properties may be inherited.

`font-family (DSSSL:font-family-name, CSS:-same-)`

Defines the name of a font/typeface family.

CSS

is an array of the following:

family-name

[Ednote: -CSS definition-]

generic-name

is one of the following:

`serif`

[Ednote: -CSS definition-]

`sans-serif`

[Ednote: -CSS definition-]

cursive

[Ednote: -CSS definition-]

fantasy

[Ednote: Where did this come from, usually "Decorative"]

monospace

A monospace font.

This is a ordered list of typeface family names. The user-agent chooses the first in the list that is available. If no match is found, uses Panose mapping to find the closest match, (again) using this list in order.

`font-style (DSSSL:font-posture, CSS:-same-)`

specifies the posture *property* of the desired font resource.

normal

Intuitively obvious.

oblique

If the face is available in oblique, you get oblique; if not but italic is available you DO NOT GET ITALIC, instead the font match fails and you attempt the next entry in the font-family list.

italic

If the face is available in italic, you get italic; if not, then if available in oblique use oblique. If neither is available the font match fails and you attempt the next entry in the font-family list.

The initial value is *normal*.

`font-style-math (DSSSL:font-posture-math, CSS:-none-) [defer]`

specifies the posture *qualifier* of the desired font resource to be used when the has the value *math*. It shall have the value *any* or one of the symbols.

The initial value is the value of the `font-style-math` character *qualifier* of the `char` *property*.

`font-stretch (DSSSL:font-proportionate-width, CSS:-same-)`

Specifies the designed setwidth (aspect ratio) of the desired member of the font-family.

is one of the following:

narrower

[Ednote: -CSS definition-]

wider

Sets font stretch to the next value above

ultra-condensed

[Ednote: -CSS definition-]

extra-condensed

[Ednote: -CSS definition-]

condensed

[Ednote: -CSS definition-]

semi-condensed

[Ednote: -CSS definition-]

normal

[Ednote: -CSS definition-]

semi-expanded

[Ednote: -CSS definition-]

expanded

[Ednote: -CSS definition-]

extra-expanded

[Ednote: -CSS definition-]

ultra-expanded

[Ednote: -CSS definition-]

The initial value is *`normal`*.

`font-size (DSSSL:-same-, CSS:-same-)`

Specifies the "body" height of the typeface. For Roman/Latin fonts, this is measured as the height from the bottom of the lowest descender to the top of the tallest ascender or highest accent in the typeface.

Can be specified using any of the following:

an absolute-size

[Ednote: -CSS definition-]

a relative-size

Size specified in EMs or EXs, relative to inherited size.

a length-specifier

[Ednote: -CSS definition-]

a percentage

Size specified in percent, relative to inherited size. [Ednote: ?]

The initial value is [Ednote: -TBD-].

`font-size-adjust (DSSSL:-none-, CSS:-same-)`

Specifies an adjustment to the "body" height of the typeface.

- CSS:

  is one of the following:

  number

  [Ednote: -CSS definition-]

  `none`

  [Ednote: -CSS definition-]

  The initial value is [Ednote: -CSS definition-].

`font-variant (DSSSL:-none-, CSS:-same-)`

Specifies if lowercase letters are to be normal or small caps in the desired font resource.

Is one of the following:

`normal`

Use standard lowercase representation.

`small-caps`

Use small caps if available or synthesize.

The initial value is *`normal`*.

`font-weight (DSSSL:-same-, CSS:-same-)`

Specifies the weight *property* of the desired font resource.

[Ednote: Neither system adequately covers the full range of 40+ weight designators in common usage for Roman/Latin fonts.]

Is one of the following:

`normal`

Same as*`400`*.

`bold`

Same as *`700`*.

`bolder`

[Ednote: -CSS definition-]

`lighter`

[Ednote: -CSS definition-]

a number between 100 & 900

Note that this is a list of defined numbers, not a numeric value that can be anywhere in the range.

[Ednote: -CSS definition-]

There is no correlation between a setting of 700 in one font and an identical value in another. The value scales are unique to each font.

The initial value is [Ednote: -TBD-].

`footer-overflow (DSSSL:-none-, CSS:overflow)`

Specifies the overflow behavior for the footer area.

(See `overflow`)

`footer-precedence (DSSSL:-none-, CSS:-none)`

A *boolean*.

A value of *`true`* specifies that the footer takes precedence and extends across the body and any start-side, start-side separation, end-side and end-side-separation.

A value of *`false`* specifies that the sides take precedence over the footer. The footer has the same width as the body and any start-side or end-side extends across the height of the footer and the footer separation.

`footer-separation (DSSSL:footer-margin, CSS:-none-)`

is a *length-specifier*. Specifies the distance between the bottom of the body area to the top of the footer area.

This *property* may be inherited.

The initial value is *`18.0pt`*.

`footer-size (DSSSL:-none-, CSS:-none-)`

A length-specifier.

Specifies the height of the footer area. If the corresponding **`queue`** content is absent, this space will still be reserved.

`footer-writing-mode (DSSSL:-none-, CSS:-none-)`

a *writing-mode-specifier* or the value *`use-page-writing-mode`*.

Specifies the *writing-mode* within the footer area of a **`simple-page-master`**.

This *property* is inherited by the footer-area's children as `writing-mode`.

The initial value is *`use-page-writing-mode`*.

`graphic-line-thickness (DSSSL:line-thickness)`

is a *length-specifier* that specifies the thickness of the *graphic line* or *graphic-line*s.

[Ednote: Coordinate with W3C-VectorGraphics-WG]

This property may be inherited.

The initial value is *`1.0pt`*.

`graphic-line-offset (DSSSL:line-position)`

is a *length-specifier* that specifies the offset distance from the alignment-line to the *graphic line* or *graphic-line*s.

[Ednote: Coordinate with W3C-VectorGraphics-WG]

This property may be inherited.

The initial value is *`1.0pt`*.

`graphic-max-width`

is a *length-specifier*. Specifies the maximum allowed width of the resulting *area* when `scale` is *`max`* or *`max-uniform`*.

This *property* is not inherited.

`graphic-max-height`

is a *length-specifier*. Specifies the maximum allowed height of the resulting *area* when `scale` is *`max`* or *`max-uniform`*.

This *property* is not inherited.

`graphic-notation-id`

An *id-specifier*.

Specifies the system mime-type of the notation of the graphic.

This *property* is not inherited and shall be specified.

`graphic-scale`

is one of the following:

a number

the graphic shall be scaled by that factor in both the horizontal and vertical directions.

a list of two numbers

the graphic shall be scaled by the factor specified by the first *number* in the horizontal direction and by the factor specified by the second *number* in the vertical direction.

`max`

If it is the symbol `max`, then it shall be scaled in the horizontal and vertical directions so that its size in the horizontal and vertical directions is as large as allowed.

`max-uniform`.

If it is the symbol `max-uniform`, then it shall be scaled uniformly in the horizontal and vertical directions so that its size in either the horizontal or vertical direction is as large as allowed.

This *property* is not inherited.

The default value is `max-uniform`.

`header-overflow (DSSSL:-none-, CSS:overflow)`

Specifies the overflow behavior for the header area.

(See `overflow`)

`header-precedence (DSSSL:-none-, CSS:-none)`

A *boolean*.

A value of `true` specifies that the header takes precedence and extends across the body and any start-side, start-side separation, end-side and end-side-seapartion.

A value of `false` specifies that the sides take precedence over the header. The header has the same width as the body and any start-side or end-side extends across the height of the header and the header separation..

`header-separation (DSSSL:header-margin, CSS:-none-)`

is a *length-specifier*. Specifies the distance between the top of the body area to the adjacent header area.

This *property* may be inherited.

The initial value is `18.0pt`.

`header-size (DSSSL:-none-, CSS:-none-)`

A length-specifier.

Specifies the height of the header area. If the corresponding **`queue`** content is absent, this space will still be reserved.

`header-writing-mode (DSSSL:-none-, CSS:-none-)`

a *writing-mode-specifier* or the value *`use-page-writing-mode`*.

Specifies the *writing-mode* within the header area of a **`simple-page-master`**.

This *property* is inherited by the header-area's children as `writing-mode`.

The initial value is *`use-page-writing-mode`*.

`height`

is a *length-specifier*. Specifies the height of the *area*.

This *property* is not inherited.

`id`

An *id-specifier*.

Specifies a unique identifier for this object within all members of the *formatter-object-tree*.

Usually optional, default=*`none`*.

Required if this object must be referenced by another object in the *formatter-object-tree*.

`indent-end (DSSSL:end-indent, CSS:-object-margin-)`

is a *length-specifier*.

Specifies the indent of the ending edge of the area in the direction of the `inline-progression-direction`.

This property may be inherited.

The initial value is *`0.0pt`*.

`indent-start (DSSSL:start-indent, CSS:-object-margin-)`

is a *length-specifier*.

Specifies the indent of the starting edge of the area in the direction of the `inline-progression-direction`.

This property may be inherited.

The initial value is *`0.0pt`*.

`indent-first-line-start (DSSSL:first-line-start-indent, CSS:text-indent)`

is a *length-specifier* giving an indent to be added to the `indent-start` for the first *textline*. This *length-specifier* may be negative. (User agents may limit a negative indent-first-line-start to a value less-than-or-equal to the applicable value of `indent-start`.)

This property may be inherited.

The initial value is *`0.0pt`*.

`inhibit-wrap`

is a *boolean*.

Specifies whether *textline* breaks shall be inhibited before and after each *area* produced by this *formatting object*. This applies only to *textline* breaks introduced by the formatter to make *textline*s fit in the available space.

The default value is *`false`*.

`letterspacing-after-maximum (DSSSL:inline-space-after, CSS:letter-spacing)`

A *length-specifier*.

This value indicates the greatest amount of inter-character space to be added to each letter.

The value may be negative (indicating the amount of space to remove). There may be implementation specific limits on the length specified.

- DSSSL:

  is an *length-specifier*.

  Specifies the minimum-space, optimum-space, and maximum-space to be added after the last result *area* in the *inline-progression-direction*.

- CSS:

  is one of the following:

  `normal`

Use the normal spacing from the font as an initial value. The formatter may add or remove space to justify the line.

a length-specifier

This value indicates the amount of inter-character space to be added to each letter. The value may be negative (indicating the amount of space to remove). There may be implementation specific limits on the length specified. The application may NOT adjust inter-character spacing to justify the line. (Note that a length of *0* disables automatic letterspacing.)

The default value is *0.0pt*.

`letterspacing-after-minimum (DSSSL:inline-space-after, CSS:letter-spacing)`

A *length-specifier*.

This value indicates the smallest amount of inter-character space to be added to each letter.

The value may be negative (indicating the amount of space to remove). There may be implementation specific limits on the length specified.

The default value is *0.0pt*.

`letterspacing-after-optimum (DSSSL:inline-space-after, CSS:letter-spacing)`

A *length-specifier*.

This value indicates the desired amount of inter-character space to be added to each letter.

The value may be negative (indicating the amount of space to remove). There may be implementation specific limits on the length specified.

The default value is *0.0pt*.

`wordspacing-maximum (DSSSL:inline-space-space, CSS:word-spacing)`

- DSSSL:

  is an *length-specifier* which is applicable to the *formatting object* if it is a space. This is in addition to any space from the `letterspacing-before-minimum` and `letterspacing-after-minimum` *properties*.

- CSS:

  is one of the following:

  normal

Use the normal spacing from the font as an initial value. The formatter may add or remove space to justify the line.

a length-specifier

This value indicates the amount of inter-word space to be added to each the normal space between words. The value may be negative (indicating the amount of space to remove). There may be implementation specific limits on the length specified. The application may also adjust inter-word spacing to justify the line.

The default value is *`0.0pt`*.

`wordspacing-minimum (DSSSL:inline-space-space, CSS:word-spacing)`

- DSSSL:

  is an *length-specifier* which is applicable to the *formatting object* if it is a space. This is in addition to any space from the `letterspacing-before-minimum` and `letterspacing-after-minimum` *properties*.

- CSS:

  is one of the following:

  `normal`

  Use the normal spacing from the font as an initial value. The formatter may add or remove space to justify the line.

  a length-specifier

  This value indicates the amount of inter-word space to be added to each the normal space between words. The value may be negative (indicating the amount of space to remove). There may be implementation specific limits on the length specified. The application may also adjust inter-word spacing to justify the line.

The default value is *`0.0pt`*.

`wordspacing-optimum (DSSSL:inline-space-space, CSS:word-spacing)`

- DSSSL:

  is an *length-specifier* which is applicable to the *formatting object* if it is a space. This is in addition to any space from the `letterspacing-before-minimum` and `letterspacing-after-minimum` *properties*.

- CSS:

  is one of the following:

normal

Use the normal spacing from the font as an initial value. The formatter may add or remove space to justify the line.

a length-specifier

This value indicates the amount of inter-word space to be added to each the normal space between words. The value may be negative (indicating the amount of space to remove). There may be implementation specific limits on the length specified. The application may also adjust inter-word spacing to justify the line.

The default value is *0.0pt*.

`input-record-end-ignore (DSSSL:ignore-record-end)`

is a *boolean*. Specifies whether a record-end shall be ignored. If this *property* is *true*, then a character with the `char-is-record-end` *qualifier* *true* shall be ignored.

This *property* may be inherited.

The initial value is *false*.

`input-tab`

is a *boolean*.

Specifies whether the *formatting object* is a tab on input.

This *property* is not inherited.

Characters that are tabs shall be treated differently by **block**s for which the `input-tab-expand` *qualifier* is not *none*.

The default value is the value of the `input-tab` character *qualifier* of the `char` *property* if the `char` *property* was not explicitly specified, and otherwise *none*.

`input-tab-expand (DSSSL:expand-tabs)`

is either *none* or a *strictly positive integer*. Specifies the tab interval. When a tab interval is specified, each character that has the `input-tab` *property* *true* shall be treated as equivalent to the smallest strictly positive number of spaces that when added to the number of characters following the last preceding record-end shall be a multiple of the tab interval.

This *property* may be inherited.

The initial value is *8*.

`input-whitespace`

is a *boolean*. Specifies whether the character shall be considered as whitespace on input.

This *property* is not inherited.

The default value is the value of the `input-whitespace` character *qualifier* of the `char` *property* if the `char` *property* was not explicitly specified, and otherwise *`false`*.

`input-whitespace-treatment`

is one of the following symbols:

`preserve`

Specifies no special action.

`collapse`

Specifies that a character for which the `input-whitespace` *property* is *`true`* shall be ignored if the preceding character also has the `input-whitespace` *property* *`true`*.

`ignore`

Specifies that any character for which the `input-whitespace` *property* is *`true`* shall be ignored.

The default value is *`preserve`*.

`keep (DSSSL:-same-)`

is one of the following:

`no-break`

the *area*s produced by this *formatting object* shall be kept together within the smallest possible *area*.

`page`

Specifies that the *area*s produced by the *formatting object* shall lie within the same page; in this case, the *formatting object* shall have an ancestor *formatting object* of class **`page-master`**.

`column-set`

Specifies that the *area*s produced by the *formatting object* shall lie within the same column set; in this case, the *formatting object* shall have an ancestor of class **`column-set`**.

column

Specifies that the *area*s produced by the *formatting object* shall lie within the same column set, and that the first column that each *area* spans in the column set shall be the same.

auto

Specifies that this *property* is to be ignored.

This *property* is not inherited.

The default value is *auto*.

orphans (DSSSL:orphan-count, CSS:orphans)

is a *positive integer*. Specifies the minimum number of *textline*s of the **block** that shall be kept together at the end of an *area*.

If the orphans is *n*, then no break shall be allowed between the first *n* *textline*s of the **block**.

This property may be inherited.

The initial value is *2*.

widows (DSSSL:widow-count, CSS:widows)

is a *positive integer*. Specifies the minimum number of *textline*s of the **block** that shall be kept together at the beginning of an *area*.

If the widows is *n*, then no break shall be allowed between the last *n* *textline*s of the **block**.

This property may be inherited.

The initial value is *2*.

keep-with-next (DSSSL:-same-)

is a *boolean*. Specifies whether the *formatting object* shall be kept in the same *area* as the next *formatting object*.

This *property* is not inherited.

The default value is *false*.

keep-with-previous (DSSSL:-same-)

is a *boolean*. Specifies whether the *formatting object* shall be kept in the same *area* as the previous *formatting object*.

This *property* is not inherited.

The default value is *`false`*.

`label-alignment`

is one of the symbols *`start`*, *`end`*, or *`center`*. Specifies the alignment of the contents of the field.

The default value is *`start`*.

`label-width`

is a *length-specifier*. Specifies the width of the *area* produced by the *formatting object*.

The default value is *`0.0pt`*.

`language (DSSSL:language & D:country)`

Has the values:

`use-document`

Specifies one should use the language/country/script specified in the source document's xml:lang specifier.

`none`

disables hyphenation and forces a simple line-breaking strategy. Used for program text and poetry.

an explicit value

has the same form as the xml:lang specifier and overrides the language derived through xml:lang.

The `language` *property* is used to control spelling, hyphenation, line-breaking, and justification. This affects *textline* composition in a system-dependent way.

This property may be inherited.

The initial value is *`use-document`*.

For reference: `country (DSSSL:country)` is *`none`* or a symbol. Specifies the country code in uppercase.

`linespacing (DSSSL:line-spacing)`

is a *length-specifier* giving the normal spacing between the *placement-path*s of *textline*s in the

`block` as described in .

This property may be inherited.

The initial value is *`12.0pt`*.

`margin-bottom`

is a *length-specifier*.

Specifies the width of the unprinted area measured inward from the bottom edge of any area.

Specifies the distance from the bottom of the page to the bottom of the lowest *area* (usually footer) used for the content in the **`simple-page-master`**.

The default value is *`0.0pt`*.

`margin-end`

is a *length-specifier*.

Specifies the width of the unprinted area measured inward from the end edge of any area.

Specifies the distance from the edge of the resulting *area* that is last in the `block-progression-direction`'s *block-progression-direction* to the nearest edge of the text *area*.

The default value is *`0.0pt`*.

`margin-left`

is a *length-specifier*.

Specifies the width of the unprinted area measured inward from the left edge of any area.

Specifies the distance from the left edge of the page to the edge of the leftmost *area* (usually start-side area) used for the content in the **`simple-page-master`**.

The default value is *`0.0pt`*.

`margin-right`

is a *length-specifier*.

Specifies the width of the unprinted area measured inward from the right edge of any area.

Specifies the distance from the right edge of the page to the edge of the rightmost *area* (usually end-side area) used for the content in the **`simple-page-master`**.

The default value is `0.0pt`.

`margin-start`

is a *length-specifier*.

Specifies the width of the unprinted area measured inward from the start edge of any area.

Specifies the distance from the edge of the resulting *area* that is first in the `block-progression-direction`'s *block-progression-direction* to the nearest edge of the text *area*.

The default value is `0.0pt`.

`margin-top`

is a *length-specifier*.

Specifies the width of the unprinted area measured inward from the top edge of any area.

Specifies the distance from the bottom of the page to the top of the highest *area* (usually header) used for the content in the **`simple-page-master`**.

The default value is `0.0pt`.

`overflow`

Specifies the action to be taken if the content of the *area* does not fit within the dimensions specified for the *area*.

Is one of the following:

`visible`

Get CSS definition.

`hidden`

Get CSS definition.

`scroll`

Get CSS definition.

`auto`

Get CSS definition.

The default value is `auto`.

`page-height (DSSSL:-same-, CSS:height [of a page-box])`

is:

- a *length-specifier*. Specifies the total height of the page (the distance between the top edge of the page to the bottom edge of the page). -or-
- *`auto`* Specifies the formatter shall determine the page height from the paper or window height.

This *property* is may be inherited.

The initial value is *`auto`*.

`page-width (DSSSL:page-width, CSS:width [of a page-box])`

is:

- a *length-specifier*. Specifies the total width of the page (the distance between the left edge of the page to the right edge of the page). -or-
- *`auto`* Specifies the formatter shall determine the page width from the paper (trim size) or window size.

This *property* is may be inherited.

The initial value is *`auto`*.

`page-writing-mode (DSSSL:-none-, CSS:-none-)`

a *writing-mode-specifier*.

Specifies the writing-mode and layout directions for a page in a **`simple-page-master`**.

This *property* is not inherited by the **`simple-page-master`**'s children, but may be accessed, where so indicated, through the *`use-page-writing-mode`* value of the `writing-mode` property.

The initial value is *`lr-tb`*.

`position-point-x`

is a *length-specifier* giving the x-coordinate of the *position-point* of the resulting *area* in the containing *area*'s coordinate system.

This applies only when the *formatting object* is *`inline`*.

This *property* is not inherited.

If this *property* is not specified and the `writing-mode` *property* is *`left-to-right`* or *`right-to-left`*, then the value shall default to *`0`*.

`position-point-y`

is a *length-specifier* giving the y-coordinate of the *position-point* of the resulting *area* in the containing *area*'s coordinate system.

This applies only when the *formatting object* is *`inline`*.

This *property* is not inherited.

If this *property* is not specified and the `writing-mode` *property* is *`top-to-bottom`*, then the value shall default to *`0`*.

`position-point-shift`

is a *length-specifier*. Specifies a shift of the *position-point* in the *shift-direction*.

The default value is *`0.0pt`*.

`queue-name (DSSSL:-none-, CSS:-none-)`

A *name-specifier*.

Defines the name of this queue.

`rule-graphic-length`

is a *length-specifier*. Specifies the length of the **`rule-graphic`**.

This *property* is not inherited. If this *property* is not specified, the *length* of the **`rule-graphic`** shall be determined by the context in which it is used.

`rule-graphic-orientation`

is one of the symbols

- *`horizontal`*,
- *`vertical`*,
- *`escapement`*, or
- *`textline-placement`*

which specifies the orientation of the `rule-graphic` and also determines whether the `rule-graphic` is *`inline`* or *`block-level`*.

1. If the orientation is *`horizontal`* or *`vertical`*, then the `rule-graphic` is *`block-level`*.

   In this case:

   - If the orientation of the `rule-graphic` is perpendicular to the *line-progression-direction*, then the size of the *area* in the *line-progression-direction* shall be 0;

   - If the orientation of the `rule-graphic` is parallel to the *line-progression-direction*, the size of the *area* in the *line-progression-direction* shall be equal to the *length* of the `rule-graphic`.

     NOTE: The size of the *area* is distinct from the thickness of the `rule-graphic`.

2. If the orientation is *`escapement`*, then the `rule-graphic` shall be *`inline`*.

   In this case, the `rule-graphic` shall be centered in the *shift-direction* about the *position-point*, and the escapement shall be equal to the *length* of the `rule-graphic`. The `rule-graphic` may be offset in the *shift-direction* using the `position-point-shift` *property*.

3. If the orientation is *`textline-placement`*, the `rule-graphic` shall be *`inline`*.

   In this case, the `rule-graphic` shall start at the *position-point* and extend in the *shift-direction* the *length* of the `rule-graphic`.

   The escapement shall be *`0`*.

   NOTE: Thus, a `rule-graphic` whose orientation is *`textline-placement`* does not affect the positioning of subsequent *formatting object*s.

This *property* is not inherited.

It has no default value and so it shall be specified.

`scale`

is a *number*. Specifies a scaling factor to be applied to the content of the *area*. *Number*s less than *`1`* shall make the content smaller. *Number*s greater than *`1`* shall make it larger.

This *property* is not inherited.

If not specified, it shall default to *`1`*.

`score-spaces`

is a *boolean*. Specifies whether the scoring shall be applied to spaces.

The default value is `true`.

`space-after-maximum (DSSSL:space-after)`

is a *length-specifier*.

Specifies the maximum-space to be inserted after the *area*s produced by the *formatting object* in the *block-progression-direction*.

This *property* is not inherited.

The default is for `no space after` to be inserted.

Non-core

`space-after-minimum (DSSSL:space-after)`

is a *length-specifier*.

Specifies the minimum-space to be inserted after the *area*s produced by the *formatting object* in the *block-progression-direction*.

This *property* is not inherited.

The default is for `no space after` to be inserted.

Non-core

`space-after-optimum (DSSSL:space-after)`

is a *length-specifier*.

Specifies the optimum-space to be inserted after the *area*s produced by the *formatting object* in the *block-progression-direction*.

This *property* is not inherited.

The default is for `no space after` to be inserted.

`space-before-maximum (DSSSL:space-before)`

is a *length-specifier*.

Specifies the maximum-space to be inserted before the *area*s produced by the *formatting object* in the *block-progression-direction*.This property is not inherited.

The default is for `no space before` to be inserted.

Non-core

`space-before-minimum (DSSSL:space-before)`

is a *length-specifier*.

Specifies the minimum-space to be inserted before the *area*s produced by the *formatting object* in the *block-progression-direction*.This property is not inherited.

The default is for *`no space before`* to be inserted.

Non-core

`space-before-optimum (DSSSL:space-before)`

is a *length-specifier*.

Specifies the optimum-space to be inserted before the *area*s produced by the *formatting object* in the *block-progression-direction*.This property is not inherited.

The default is for *`no space before`* to be inserted.

`start-side-overflow (DSSSL:-none-, CSS:overflow)`

Specifies the overflow behavior for the start-side area.

(See `overflow`)

`start-side-separation (DSSSL:footer-margin, CSS:-none-)`

is a *length-specifier*. Specifies the distance from the edge of the body area to the adjacent start-side area.

This *property* may be inherited.

The initial value is *`18.0pt`*.

`start-side-size (DSSSL:-none-, CSS:-none-)`

A length-specifier.

Specifies the width of the start-side area. If the corresponding **`queue`** content is absent, this space will still be reserved.

`start-side-writing-mode (DSSSL:-none-, CSS:-none-)`

a *writing-mode-specifier* or the value *`use-page-writing-mode`*.

Specifies the *writing-mode* within the start-side area of a **`simple-page-master`**.

This *property* is inherited by the start-side-area's children as `writing-mode`.

The initial value is *`use-page-writing-mode`*.

`master-name (DSSSL:-none-, CSS:-none-)`

A *name-specifier*.

Specifies the name of a **`simple-page-master`**. Used in the **`page-sequence`**'s `sequence-rule` to control activation of this master.

For this draft, `master-name`s are restricted to the values: *`"first"`*, *`"odd"`*, *`"even"`* or *`"scrolling"`*.

`text-align (DSSSL:quadding, CSS:text-align)`

- DSSSL:

  is one of the symbols:

  `start`

  [Ednote: -TBD-]

  `end`

  [Ednote: -TBD-]

  `left`

  [Ednote: For CSS compatibility, -TBD-]

  `right`

  [Ednote: For CSS compatibility, -TBD-]

  `center`

  [Ednote: -TBD-]

  `justify`

  [Ednote: -TBD-]

  `justify-force`

  [Ednote: -TBD-]

Specifies the alignment of *textline*s other than the last *textline* in the **`block`** in the *line-progression-direction* determined by the `writing-mode`. A value of *`spread-inside`* or *`spread-outside`* shall be allowed only if the *formatting object* has an ancestor of class **`page-master`**. A value of *`page-inside`* or *`page-outside`* shall be allowed only if the *formatting object* has an ancestor of **`column-set-master`**.

- CSS:

  is one of the symbols:

  `left`

  Specifies that all lines in the paragraph are left-aligned (ragged-right).

  `center`

  Specifies that all lines in the paragraph are centered.

  `right`

  Specifies that all lines in the paragraph are right-aligned (ragged-left).

  `justify`

  Specifies that all lines in the paragraph are justified (spread), except the last/only line which is left-aligned.

This property may be inherited.

The initial value is *`start`*.

`text-align-last (DSSSL:last-line-quadding, CSS:-none-)`

- DSSSL:

  is one of the symbols:

  `auto`

  Alignment of the last line in a block shall be the same as `text-align`, except if `text-align` is *`justify`* or *`justify-force`*, then the alignment of the last line shall be *`start`*.

  `start`

  [Ednote: -TBD-]

  `end`

[Ednote: -TBD-]

left

[Ednote: For CSS compatibility, -TBD-]

right

[Ednote: For CSS compatibility, -TBD-]

center

[Ednote: -TBD-]

justify

[Ednote: -TBD-]

justify-force

[Ednote: -TBD-]

Specifies the alignment of the last *textline* in the **block** in the *line-progression-direction* determined by the `writing-mode`.

A value of *auto* specifies that the value of the `text-align` *property* shall be used, except when that value is *justify* or *justify-force*, in which case, a value of *start* shall be used.

A value of *spread-inside* or *spread-outside* shall be allowed only if the *formatting object* has an ancestor of class **page-master**. A value of *page-inside* or *page-outside* shall be allowed only if the *formatting object* has an ancestor of **column-set-master**.

- CSS: assumes:*left*

This property may be inherited.

The initial value is *auto*.

Non-core

`block-justification-letterspace-max-add (DSSSL:justify-glyph-space-max-add)`

is a *length-specifier*. Specifies the maximum space that may be added between glyphs in order to justify a *textline*.

NOTE: The interaction between `block-justification-letterspace-max-add`, `block-justification-letterspace-max-remove`, `block-justification-wordspace-max`, and `block-justification-wordspace-min` is system dependent.

This property may be inherited.

The initial value is *`0.0pt`*.

`block-justification-letterspace-max-remove (DSSSL:justify-glyph-space-max-remove)`

is a *length-specifier*. Specifies the maximum space that may be removed between glyphs in order to justify a *textline*.

> **NOTE:** The interaction between `block-justification-letterspace-max-add`, `block-justification-letterspace-max-remove`, `block-justification-wordspace-max`, and `block-justification-wordspace-min` is system dependent.

This property may be inherited.

The initial value is *`0.0pt`*.

`block-justification-wordspace-max (DSSSL:-none-)`

is a *number* from 100 to 400 indicating the percent of the nominal wordspace width. Specifies the maximum width that the wordspaces may have in order to justify a *textline*.

> **NOTE:** The interaction between `block-justification-letterspace-max-add`, `block-justification-letterspace-max-remove`, `block-justification-wordspace-max`, and `block-justification-wordspace-min` is system dependent.

This property may be inherited.

The initial value is *`200.0%`*.

`block-justification-wordspace-min (DSSSL:-none-)`

is a *number* from 0 to 100 indicating the percent of the nominal wordspace width. Specifies the minimum width that the wordspaces may have in order to justify a *textline*.

> **NOTE:** The interaction between `block-justification-letterspace-max-add`, `block-justification-letterspace-max-remove`, `block-justification-wordspace-max`, and `block-justification-wordspace-min` is system dependent.

This property may be inherited.

The initial value is *`50.0%`*.

`block-line-breaking (DSSSL:lines)`

is a symbol. Specifies how the content of the **`block`** shall be broken into *textline*s in the formatted output, as follows:

`wrap`

Specifies that *textline*s shall be broken so that they fit in the available space.

`asis`

Specifies that *textline*s shall be broken only after characters for which the `char-is-record-end` *property* is *`true`*.

`asis-wrap`

Specifies that *textline*s shall be broken after characters for which the `char-is-record-end` *property* is *`true`*, and as necessary to make *textline*s fit in the available space.

`asis-truncate`

Specifies that *textline*s shall be broken only after characters for which the `char-is-record-end` *property* is *`true`*, and that *textline*s that do not fit the in the available space shall be truncated.

`none`

Specifies that *textline*s shall not be broken at all.

> **NOTE:** This is useful in **`table`**s when the `table-width` property is *`auto`* to ensure that the width of a column is made large enough so that the content of a cell fits on a single *textline*.

In all cases, *textline* breaks shall also be allowed where explicitly specified with the `break-before` or `break-after` *properties*.

This property may be inherited.

The initial value is *`wrap`*.

`width`

is a *length-specifier*. Specifies the width of the *area*.

This *property* is not inherited.

`writing-mode (DSSSL:-same-)`

a *writing-mode-specifier*.

May be specified and is used by most text and layout objects.

This *property* may be inherited.

The initial value is `left-to-right`.

A. DTD for XSL Stylesheets

The following entity can be used to construct a DTD for XSL 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:sequence
  | fo:block
">
```

The stylesheet DTD may also need to define additional attributes for `xsl:attribute-set`.

```
<!ENTITY % instructions "
  | xsl:process-children
  | xsl:process
  | xsl:for-each
  | xsl:value-of
  | xsl:number
  | xsl:choose
  | xsl:if
  | xsl:contents
  | xsl:invoke
  | xsl:text
">

<!ENTITY % template "
 (#PCDATA
  %instructions;
  %result-elements;)*
">

<!ENTITY % space-att "xml:space (default|preserve) #IMPLIED">

<!ELEMENT xsl:stylesheet
 (xsl:import*,
  (xsl:include
  | xsl:id
  | xsl:strip-space
  | xsl:preserve-space
  | xsl:define-macro
  | xsl:define-attribute-set
  | xsl:define-constant
  | xsl:template)*)
>

<!ATTLIST xsl:stylesheet
  result-ns NMTOKEN #IMPLIED
```

```
  default-space (preserve|strip) "preserve"
  indent-result (yes|no) "no"
  id ID #IMPLIED
  xmlns:xsl CDATA #FIXED "http://www.w3.org/TR/WD-xsl"
  %space-att;
>

<!-- Used for attribute values that are URIs.-->
<!ENTITY % URI "CDATA">

<!-- Used for attribute values that are patterns.-->
<!ENTITY % pattern "CDATA">

<!-- Used for attribute values that are a priority. -->
<!ENTITY % priority "NMTOKEN">

<!ELEMENT xsl:import EMPTY>
<!ATTLIST xsl:import href %URI; #REQUIRED>

<!ELEMENT xsl:include EMPTY>
<!ATTLIST xsl:include href %URI; #REQUIRED>

<!ELEMENT xsl:id EMPTY>
<!ATTLIST xsl:id
  attribute NMTOKEN #REQUIRED
  element NMTOKEN #IMPLIED
>

<!ELEMENT xsl:strip-space EMPTY>
<!ATTLIST xsl:strip-space element NMTOKEN #REQUIRED>

<!ELEMENT xsl:preserve-space EMPTY>
<!ATTLIST xsl:preserve-space element NMTOKEN #REQUIRED>

<!ELEMENT xsl:template %template;>
<!ATTLIST xsl:template
  match %pattern; #REQUIRED
  priority %priority; #IMPLIED
  %space-att;
>

<!ELEMENT xsl:attribute-set EMPTY>

<!ATTLIST xsl:attribute-set
  xsl:use NMTOKENS #IMPLIED
>

<!ELEMENT xsl:process-children EMPTY>

<!ELEMENT xsl:value-of EMPTY>
<!ATTLIST xsl:value-of expr CDATA #IMPLIED>

<!ENTITY % conversion-atts '
   format CDATA "1"
   xml:lang NMTOKEN #IMPLIED
   letter-value (alphabetic|other) #IMPLIED
   digit-group-sep CDATA #IMPLIED
   n-digits-per-group NMTOKEN #IMPLIED
   sequence-src %URI; #IMPLIED
'>
```

```
<!ELEMENT xsl:number EMPTY>
<!ATTLIST xsl:number
   level (single|multi|any) "single"
   count CDATA #IMPLIED
   from CDATA #IMPLIED
   %conversion-atts;
>

<!ELEMENT xsl:process EMPTY>
<!ATTLIST xsl:process
  select %pattern; #REQUIRED
>

<!ELEMENT xsl:for-each %template;>
<!ATTLIST xsl:for-each
  select %pattern; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:if %template;>
<!ATTLIST xsl:if
  test %pattern; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:choose (xsl:when+, xsl:otherwise?)>
<!ATTLIST xsl:choose %space-att;>

<!ELEMENT xsl:when %template;>
<!ATTLIST xsl:when
  test %pattern; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:otherwise %template;>
<!ATTLIST xsl:otherwise %space-att;>

<!ELEMENT xsl:define-attribute-set (xsl:attribute-set)>
<!ATTLIST xsl:define-attribute-set
  name NMTOKEN #REQUIRED
>

<!ELEMENT xsl:define-constant EMPTY>
<!ATTLIST xsl:define-constant
  name NMTOKEN #REQUIRED
  value CDATA #REQUIRED
>

<!-- xsl:macro-arg cannot occur after any other elements or
any non-whitespace character -->

<!ELEMENT xsl:define-macro
 (#PCDATA
  %instructions;
  %result-elements;
  | xsl:macro-arg)*
>

<!ATTLIST xsl:define-macro
  name NMTOKEN #REQUIRED
  %space-att;
```

```
>

<!ELEMENT xsl:macro-arg EMPTY>
<!ATTLIST xsl:macro-arg
  name NMTOKEN #REQUIRED
  default CDATA #IMPLIED
>

<!-- This is allowed only within xsl:define-macro -->
<!ELEMENT xsl:contents EMPTY>

<!-- xsl:arg cannot occur after any other elements or
any non-whitespace character -->

<!ELEMENT xsl:invoke
 (#PCDATA
  %instructions;
  %result-elements;
  | xsl:arg)*
>

<!ATTLIST xsl:invoke
  macro NMTOKEN #REQUIRED
  %space-att;
>

<!ELEMENT xsl:arg EMPTY>
<!ATTLIST xsl:arg
  name NMTOKEN #REQUIRED
  value CDATA #REQUIRED
>

<!ELEMENT xsl:text (#PCDATA)>
<!ATTLIST xsl:text %space-att;>
```

B. References

B.1 Normative References

W3C XML
: World Wide Web Consortium. *Extensible Markup Language (XML) 1.0.* W3C Recommendation. See http://www.w3.org/TR/1998/REC-xml-19980210

W3C XML Names
: World Wide Web Consortium. *Namespaces in XML.* W3C Working Draft. See http://www.w3.org/TR/WD-xml-names

B.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.

C. Examples (Non-Normative)

The following is a simple but complete stylesheet.

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl"
                xmlns:fo="http://www.w3.org/TR/WD-xsl/FO"
                result-ns="fo"
                indent-result="yes">
<xsl:template match='/'>
 <fo:page-sequence font-family="serif">
  <fo:simple-page-master name='scrolling'/>
  <fo:queue queue-name='body'>
   <xsl:process-children/>
  </fo:queue>
 </fo:page-sequence>
</xsl:template>

<xsl:template match="title">
 <fo:block font-weight="bold">
  <xsl:process-children/>
 </fo:block>
</xsl:template>

<xsl:template match="p">
 <fo:block>
  <xsl:process-children/>
 </fo:block>
</xsl:template>

<xsl:template match="emph">
 <fo:sequence font-style="italic">
  <xsl:process-children/>
 </fo:sequence>
</xsl:template>

</xsl:stylesheet>
```

With the following source document

```
<doc>
<title>An example</title>
<p>This is a test.</p>
<p>This is <emph>another</emph> test.</p>
</doc>
```

it would produce the following result

```
<fo:page-sequence xmlns:fo="http://www.w3.org/TR/WD-xsl/FO"
  font-family="serif">
<fo:simple-page-master name="scrolling"/>
<fo:queue queue-name="body">
<fo:block font-weight="bold">An example</fo:block>
<fo:block>This is a test.</fo:block>
<fo:block>This is <fo:sequence
  font-style="italic">another</fo:sequence> test.</fo:block>
</fo:queue>
```

```
</fo:page-sequence>
```

D. Design Principles (Non-Normative)

In the design of any language, trade-offs in the solution space are necessary. To aid in making these trade-offs the follow design principles were used:

- XSL should support browsing, printing, and interactive editing and design tools
- XSL should be capable of specifying presentations for traditional and Web environments
- XSL should support interaction with structured information, as well as presentation of it.
- XSL should support all kinds of structured information, including both data and documents.
- XSL should support both visual and non-visual presentations.
- XSL should be a declarative language.
- XSL should be optimized to provide simple specifications for common formatting tasks and not preclude more sophisticated formatting tasks.
- XSL should provide an extensibility mechanism
- The number of optional features in XSL should be kept to a minimum.
- XSL should provide the formatting functionality of *at least* DSSSL and CSS
- XSL should leverage other recommendations and standards, including XML, XLL, DOM, HTML and ECMAScript.
- XSL should be expressed in XML syntax.
- XSL stylesheets should be human-readable and reasonably clear.
- Terseness in XSL markup is of minimal importance.

E. Acknowledgements (Non-Normative)

The following have contributed to authoring this draft:

- Sharon Adler, Inso Corporation
- Anders Berglund, Inso Corporation
- Paul Grosso, ArborText
- Eduardo Gutentag, Sun Microsystems
- Chris Lilley, W3C
- Chris Maden, O'Reilly & Associates
- Jonathan Marsh, Microsoft Corporation
- Henry S. Thompson, University of Edinburgh
- Paul Trevithick, Bitstream
- Norman Walsh, ArborText
- Steve Zilles, Adobe