Mathematical Markup Language (MathML) Version 4.0

W3C Working Draft 19 November 2024

▼ **More details about this document**

This version:
https://www.w3.org/TR/2024/WD-mathml4-20241119/

Latest published version:
https://www.w3.org/TR/mathml4/

Latest editor's draft:
https://w3c.github.io/mathml/

History:
https://www.w3.org/standards/history/mathml4/
Commit history

Editor:
David Carlisle (NAG)

Former editors:
Patrick Ion
Robert Miner (deceased)

Feedback:
GitHub w3c/mathml (pull requests, new issue, open issues)

Latest MathML Recommendation
https://www.w3.org/TR/MathML/

Abstract

This specification defines the Mathematical Markup Language, or MathML. MathML is a markup language for describing mathematical notation and capturing both its structure and content. The goal of MathML is to enable mathematics to be served, received, and processed on the World Wide Web, just as [HTML] has enabled this functionality for text.

This specification of the markup language MathML is intended primarily for a readership consisting of those who will be developing or implementing renderers or editors using it, or software that will communicate using MathML as a protocol for input or output. It is *not* a User's Guide but rather a reference document.

MathML can be used to encode both mathematical notation and mathematical content. About thirty-eight of the MathML tags describe abstract notational structures, while another about one hundred and seventy provide a way of unambiguously specifying the intended meaning of an expression. Additional chapters discuss how the MathML content and presentation elements interact, and how MathML renderers might be implemented and should interact with browsers. Finally, this document addresses the issue of special characters used for mathematics, their handling in MathML, their presence in

While MathML is human-readable, authors typically will use equation editors, conversion programs, and other specialized software tools to generate MathML. Several versions of such MathML tools exist, both freely available software and commercial products, and more are under development.

MathML was originally specified as an XML application and most of the examples in this specification assume that syntax. Other syntaxes are possible, most notably [HTML] specifies the syntax for MathML in HTML. Unless explicitly noted, the examples in this specification are also valid HTML syntax.

Status of This Document

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

Public discussion of MathML and issues of support through the W3C for mathematics on the Web takes place on the public mailing list of the Math Working Group (list archives). To subscribe send an email to www-math-request@w3.org with the word `subscribe` in the subject line. Alternatively, report an issue at this specification's GitHub repository.

A fuller discussion of the document's evolution can be found in I. Changes.

Some sections are collapsed and may be expanded to reveal more details. The following button may be used to expand all such sections. Expand All Sections

This document was published by the Math Working Group as a Working Draft using the Recommendation track.

Publication as a Working Draft does not imply endorsement by W3C and its Members.

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

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

This document is governed by the 03 November 2023 W3C Process Document.

Issue summary

Issue 304: Potential presentation MathML items to deprecate in MathML 4

Issue 180: "decimalpoint" value definition

Issue 284: Make the sample presentation of Strict Content use intent

Issue 247: Spec should specify what char to use for accents/lines

Issue 178: Make MathML attributes ASCII case-insensitive

Issue 361: structuring common attributes

Table of Contents

1. Introduction

This section is non-normative.

1.1 Mathematics and its Notation

Mathematics and its notations have evolved over several centuries, or even millennia. To the experienced reader, mathematical notation conveys a large amount of information quickly and compactly. And yet, while the symbols and arrangements of the notations have a deep correspondence to the semantic structure and meaning of the mathematics being represented, the notation and semantics are not the same. The semantic symbols and structures are subtly distinct from those of the notation.

Thus, there is a need for a markup language which can represent both the traditional displayed notations of mathematics, as well as its semantic content. While the traditional rendering is useful to sighted readers, the markup language must also support accessibility. The semantic forms must support a variety of computational purposes. Both forms should be appropriate to all educational levels from elementary to research.

1.2 Overview

MathML is a markup language for describing mathematics. It uses XML syntax when used standalone or within other XML, or HTML syntax when used within HTML documents. Conceptually, MathML consists of two main strains of markup: Presentation markup is used to display mathematical expressions; and Content markup is used to convey mathematical meaning. These two strains, along with other external representations, can be combined using parallel markup.

This specification is organized as follows: 2. MathML Fundamentals discusses Fundamentals common to Presentation and Content markup; 3. Presentation Markup and 4. Content Markup cover Presentation and Content markup, respectively; 5. Annotating MathML: intent discusses how markup may be annotated, particularly for accessibility; 6. Annotating MathML: semantics discusses how markup may be annotated so that Presentation, Content and other formats may be combined; 7. Interactions with the Host Environment addresses how MathML interacts with applications; Finally, a discussion of special symbols, and issues regarding characters, entities and fonts, is given in 8. Characters, Entities and Fonts.

1.3 Relation to MathML Core

The specification of MathML is developed in two layers. MathML Core ([MathML-Core]) covers (most of) Presentation Markup, with the focus being the precise details of displaying mathematics in web browsers. MathML Full, this

specification, extends *MathML Core* primarily by defining Content MathML, in 4. Content Markup. It also defines extensions to Presentation MathML consisting of additional attributes, elements or enhanced syntax of attributes. These are defined for compatibility with legacy MathML, as well as to cover 3.1.7 Linebreaking of Expressions, 3.6 Elementary Math and other aspects not included in level 1 of *MathML Core* but which may be incorporated into future versions of *MathML Core*.

This specification covers both MathML Core and its extensions; features common to both are indicated with core, whereas extensions are indicated with not-core.

It is intended that MathML Full is a proper superset of MathML Core. Moreover, it is intended that any valid Core Markup be considered as valid Full Markup as well. It is also intended that an otherwise conforming implementation of MathML Core, which also implements parts or all of the extensions of MathML Full, should continue to be considered a conforming implementation of MathML Core.

1.4 MathML Notes

In addition to these two specifications, the Math WG group has developed the non-normative *Notes on MathML* that contains additional examples and information to help understand best practices when using MathML.

2. MathML Fundamentals

2.1 MathML Syntax and Grammar

2.1.1 General Considerations

The basic ‘syntax’ of MathML is defined using XML syntax, but other syntaxes that can encode labeled trees are possible. Notably the HTML parser may also be used with MathML. Upon this, we layer a ‘grammar’, being the rules for allowed elements, the order in which they can appear, and how they may be contained within each other, as well as additional syntactic rules for the values of attributes. These rules are defined by this specification, and formalized by a RelaxNG schema [RELAXNG-SCHEMA] in A. Parsing MathML. Derived schema in other formats, DTD (Document Type Definition) and XML Schema [XMLSchemas] are also provided.

MathML's character set consists of any Unicode characters [Unicode] allowed by the syntax being used. (See for example [XML] or [HTML].) The use of Unicode characters for mathematics is discussed in 8. Characters, Entities and Fonts.

The following sections discuss the general aspects of the MathML grammar as well as describe the syntaxes used for attribute values.

2.1.2 MathML and Namespaces

An XML namespace [Namespaces] is a collection of names identified by a URI. The URI for the MathML namespace is:

```
http://www.w3.org/1998/Math/MathML
```

To declare a namespace when using the XML serialisation of MathML, one uses an `xmlns` attribute, or an attribute with an `xmlns` prefix.

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <mrow>...</mrow>
</math>
```

When the `xmlns` attribute is used as a prefix, it declares a prefix which can then be used to explicitly associate other elements and attributes with a particular namespace. When embedding MathML within HTML using XML syntax, one might use:

```
<body xmlns:m="http://www.w3.org/1998/Math/MathML">
  ...
  <m:math><m:mrow>...</m:mrow></m:math>
  ...
</body>
```

HTML does not support namespace extensibility in the same way. The HTML parser has in-built knowledge of the HTML, SVG, and MathML namespaces. `xmlns` attributes are just treated as normal attributes. Thus, when using the HTML serialisation of MathML, prefixed element names must not be used. `xmlns=http://www.w3.org/1998/Math/MathML` may be used on the `math` element; it will be ignored by the HTML parser. If a MathML expression is likely to be in contexts where it may be parsed by an XML parser or an HTML parser, it *SHOULD* use the following form to ensure maximum compatibility:

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  ...
</math>
```

2.1.3 Children versus Arguments

There are presentation elements that conceptually accept only a single argument, but which for convenience have been written to accept any number of children; then we infer an `mrow` containing those children which acts as the argument to the element in question; see 3.1.3.1 Inferred <mrow>s.

In the detailed discussions of element syntax given with each element throughout the MathML specification, the number of arguments required and their order, as well as other constraints on the content, are specified. This information is also tabulated for the presentation elements in 3.1.3 Required Arguments.

2.1.4 MathML and Rendering

Web Platform implementations of [MathML-Core] should follow the detailed layout rules specified in that document.

This document only recommends (i.e., does not require) specific ways of rendering Presentation MathML; this is in order to allow for medium-dependent rendering and for implementations not using the CSS based Web Platform.

2.1.5 MathML Attribute Values

MathML elements take attributes with values that further specialize the meaning or effect of the element. Attribute names are shown in a `monospaced` font throughout this document. The meanings of attributes and their allowed values are described within the specification of each element. The syntax notation explained in this section is used in specifying allowed values.

2.1.5.1 Syntax notation used in the MathML specification

To describe the MathML-specific syntax of attribute values, the following conventions and notations are used for most attributes in the present document.

Notation	What it matches
unsigned-integer	As defined in [MathML-Core], an `integer`, whose first character is neither U+002D HYPHEN-MINUS character (-) nor U+002B PLUS SIGN (+).
positive-integer	An unsigned-integer not consisting solely of "0"s (U+0030), representing a positive integer
integer	an optional "-" (U+002D), followed by an unsigned-integer, and representing an integer
unsigned-number	value as defined in [CSS-VALUES-3] `number`, whose first character is neither U+002D HYPHEN-MINUS character (-) nor U+002B PLUS SIGN (+), representing a non-negative terminating decimal number (a type of rational number)
number	an optional prefix of "-" (U+002D), followed by an unsigned number, representing a terminating decimal number (a type of rational number)
character	a single non-whitespace character
string	an arbitrary, nonempty and finite, string of *character*s
length	a length, as explained below, 2.1.5.2 Length Valued Attributes
namedspace	a named length, namedspace, as explained in 2.1.5.2 Length Valued Attributes
color	a color, using the syntax specified by [CSS-Color-3]
id	an identifier, unique within the document; must satisfy the NAME syntax of the XML recommendation [XML]
idref	an identifier referring to another element within the document; must satisfy the NAME syntax of the XML recommendation [XML]
URI	a Uniform Resource Identifier [RFC3986]. Note that the attribute value is typed in the schema as anyURI which allows any sequence of XML characters. Systems needing to use this string as a URI must encode the bytes of the UTF-8 encoding of any characters not allowed in URI using %HH encoding where HH are the byte value in hexadecimal. This ensures that such an attribute value may be interpreted as an IRI, or more generally a LEIRI; see [IRI].
italicized word	values as explained in the text for each attribute; see 2.1.5.3 Default values of attributes
"literal"	quoted symbol, literally present in the attribute value (e.g. "+" or '+')

The ‘types’ described above, except for *string*, may be combined into composite patterns using the following operators. The whole attribute value must be delimited by single (') or double (") quotation marks in the marked up document. Note that double quotation marks are often used in this specification to mark up literal expressions; an example is the "-" in line 5 of the table above.

In the table below a form *f* means an instance of a type described in the table above. The combining operators are shown in order of precedence from highest to lowest:

Notation	What it matches
(f)	same f
$f?$	an optional instance of f
$f*$	zero or more instances of f, with separating whitespace characters
$f+$	one or more instances of f, with separating whitespace characters
$f_1 f_2 \cdots f_n$	one instance of each form f_i, in sequence, with no separating whitespace
$f_1, f_2, ..., f_n$	one instance of each form f_i, in sequence, with separating whitespace characters (but no commas)
$f_1 \mid f_2 \mid \cdots \mid f_n$	any one of the specified forms f_i

The notation we have chosen here is in the style of the syntactical notation of the RelaxNG used for MathML's basic schema, A. Parsing MathML.

Since some applications are inconsistent about normalization of whitespace, for maximum interoperability it is advisable to use only a single whitespace character for separating parts of a value. Moreover, leading and trailing whitespace in attribute values should be avoided.

For most numerical attributes, only those in a subset of the expressible values are sensible; values outside this subset are not errors, unless otherwise specified, but rather are rounded up or down (at the discretion of the renderer) to the closest value within the allowed subset. The set of allowed values may depend on the renderer, and is not specified by MathML.

If a numerical value within an attribute value syntax description is declared to allow a minus sign ('-'), e.g., `number` or `integer`, it is not a syntax error when one is provided in cases where a negative value is not sensible. Instead, the value should be handled by the processing application as described in the preceding paragraph. An explicit plus sign ('+') is not allowed as part of a numerical value except when it is specifically listed in the syntax (as a quoted '+' or "+"), and its presence can change the meaning of the attribute value (as documented with each attribute which permits it).

§ *2.1.5.2 Length Valued Attributes*

Most presentation elements have attributes that accept values representing lengths to be used for size, spacing or similar properties. [MathML-Core] accepts lengths only in the `<length-percentage>` syntax defined in [CSS-VALUES-3]. MathML Full extends length syntax by accepting also a ***namedspace*** being one of:

Positive space	Negative space	Value
veryverythinmathspace	negativeveryverythinmathspace	±1/18 em
verythinmathspace	negativeverythinmathspace	±2/18 em
thinmathspace	negativethinmathspace	±3/18 em
mediummathspace	negativemediummathspace	±4/18 em
thickmathspace	negativethickmathspace	±5/18 em
verythickmathspace	negativeverythickmathspace	±6/18 em
veryverythickmathspace	negativeveryverythickmathspace	±7/18 em

In MathML 3, the attributes on mpadded allowed three ***pseudo-units***, `height`, `depth`, and `width` (taking the place of one of the usual CSS units) denoting the original dimensions of the content. It also allowed a deprecated usage with lengths specified as a number without a unit which was interpreted as a multiple of the reference value. These forms are considered invalid in MathML 4.

2.1.5.2.1 Additional notes about units

Two additional aspects of relative units must be clarified, however. First, some elements such as 3.4 Script and Limit Schemata or `mfrac` implicitly switch to smaller font sizes for some of their arguments. Similarly, `mstyle` can be used to explicitly change the current font size. In such cases, the effective values of an `em` or `ex` inside those contexts will be different than outside. The second point is that the effective value of an `em` or `ex` used for an attribute value can be affected by changes to the current font size. Thus, attributes that affect the current font size, such as `mathsize` and `scriptlevel`, must be processed before evaluating other length valued attributes.

2.1.5.3 Default values of attributes

Default values for MathML attributes are, in general, given along with the detailed descriptions of specific elements in the text. Default values shown in plain text in the tables of attributes for an element are literal, but when italicized are descriptions of how default values can be computed.

Default values described as *inherited* are taken from the rendering environment, as described in 3.3.4 Style Change `<mstyle>`, or in some cases (which are described individually) taken from the values of other attributes of surrounding elements, or from certain parts of those values. The value used will always be one which could have been specified explicitly, had it been known; it will never depend on the content or attributes of the same element, only on its environment. (What it means when used may, however, depend on those attributes or the content.)

Default values described as *automatic* should be computed by a MathML renderer in a way which will produce a high-quality rendering; how to do this is not usually specified by the MathML specification. The value computed will always be

one which could have been specified explicitly, had it been known, but it will usually depend on the element content and possibly on the context in which the element is rendered.

Other italicized descriptions of default values which appear in the tables of attributes are explained individually for each attribute.

The single or double quotes which are required around attribute values in an XML start tag are not shown in the tables of attribute value syntax for each element, but are around attribute values in examples in the text, so that the pieces of code shown are correct.

Note that, in general, there is no mechanism in MathML to simulate the effect of not specifying attributes which are *inherited* or *automatic*. Giving the words "inherited" or "automatic" explicitly will not work, and is not generally allowed. Furthermore, the `mstyle` element (3.3.4 Style Change `<mstyle>`) can even be used to change the default values of presentation attributes for its children.

Note also that these defaults describe the behavior of MathML applications when an attribute is not supplied; they do not indicate a value that will be filled in by an XML parser, as is sometimes mandated by DTD-based specifications.

In general, there are a number of properties of MathML rendering that may be thought of as overall properties of a document, or at least of sections of a large document. Examples might be `mathsize` (the math font size: see 3.2.2 Mathematics style attributes common to token elements), or the behavior in setting limits on operators such as integrals or sums (e.g., `movablelimits` or `displaystyle`), or upon breaking formulas over lines (e.g. `linebreakstyle`); for such attributes see several elements in 3.2 Token Elements. These may be thought to be inherited from some such containing scope. Just above we have mentioned the setting of default values of MathML attributes as *inherited* or *automatic*; there is a third source of global default values for behavior in rendering MathML, a MathML operator dictionary. A default example is provided in B. Operator Dictionary. This is also discussed in 3.2.5.6.1 The operator dictionary and examples are given in 3.2.5.2.1 Dictionary-based attributes.

2.1.6 Attributes Shared by all MathML Elements

In addition to the attributes described specifically for each element, the attributes in the following table are allowed on every MathML element. Also allowed are attributes from the xml namespace, such as `xml:lang`, and attributes from namespaces other than MathML, which are ignored by default.

Name	values	default
id[core]	*id*	*none*
	Establishes a unique identifier associated with the element to support linking, cross-references and parallel markup. See `xref` and 6.9 Parallel Markup.	
xref[not-core]	*idref*	*none*
	References another element within the document. See `id` and 6.9 Parallel Markup.	
class[core]	*string*	*none*
	Associates the element with a set of style classes for use with [CSS21]. See 7.5 Using CSS with MathML for discussion of the interaction of MathML and CSS.	
style[core]	*string*	*none*
	Associates style information with the element for use with [CSS21]. See 7.5 Using CSS with MathML for discussion of the interaction of MathML and CSS.	
href[not-core]	*URI*	*none*
	Can be used to establish the element as a hyperlink to the specified *URI*.	

All MathML presentation elements accept `intent` and `arg` attributes to support specifying "intent". These are more fully described in 5. Annotating MathML: intent.

Name	values	default
intent[core]	*intent expression*	*none*
	The `intent` attribute is more fully described in 5. Annotating MathML: intent. It may be used on presentation elements to give information about the intended meaning of the expression, mainly for guiding audio or braille accessible renderings.	
arg[core]	*name*	*none*
	The `arg` attribute is more fully described in 5. Annotating MathML: intent. It may be used to name an element to be referenced from an `intent` expression on an ancestor element.	

See also 3.2.2 Mathematics style attributes common to token elements for a list of MathML attributes which can be used on most presentation token elements.

2.1.7 Collapsing Whitespace in Input

In MathML, as in XML, "whitespace" means simple spaces, tabs, newlines, or carriage returns, i.e., characters with

hexadecimal Unicode codes U+0020, U+0009, U+000A, or U+000D, respectively; see also the discussion of whitespace in Section 2.3 of [XML].

MathML ignores whitespace occurring outside token elements. Non-whitespace characters are not allowed there. Whitespace occurring within the content of token elements, except for `<cs>`, is normalized as follows. All whitespace at the beginning and end of the content is removed, and whitespace internal to content of the element is collapsed canonically, i.e., each sequence of 1 or more whitespace characters is replaced with one space character (U+0020, sometimes called a blank character).

For example, `<mo> ( </mo>` is equivalent to `<mo>(</mo>`, and

```
 <mtext>
Theorem
1:
 </mtext>
```

Theorem 1:

is equivalent to `<mtext>Theorem 1:</mtext>` or `<mtext>Theorem&#x20;1:</mtext>`.

Authors wishing to encode white space characters at the start or end of the content of a token, or in sequences other than a single space, without having them ignored, must use non-breaking space U+00A0 (or `nbsp`) or other non-marking characters that are not trimmed. For example, compare the above use of an `mtext` element with

```
 <mtext>
 <!--nbsp-->Theorem  <!--nbsp-->1:
 </mtext>
```

Theorem 1:

When the first example is rendered, there is nothing before “Theorem”, one Unicode space character between “Theorem” and “1:”, and nothing after “1:”. In the second example, a single space character is to be rendered before “Theorem”; two spaces, one a Unicode space character and one a Unicode no-break space character, are to be rendered before “1:”; and there is nothing after the “1:”.

Note that the value of the `xml:space` attribute is not relevant in this situation since XML processors pass whitespace in tokens to a MathML processor; it is the requirements of MathML processing which specify that whitespace is trimmed and collapsed.

For whitespace occurring outside the content of the token elements `mi`, `mn`, `mo`, `ms`, `mtext`, `ci`, `cn`, `cs`, `csymbol` and `annotation`, an `mspace` element should be used, as opposed to an `mtext` element containing only whitespace entities.

§ 2.2 The Top-Level `<math>` Element

MathML specifies a single top-level or root `math` element, which encapsulates each instance of MathML markup within a

document. All other MathML content must be contained in a `math` element; in other words, every valid MathML expression is wrapped in outer `<math>` tags. The `math` element must always be the outermost element in a MathML expression; it is an error for one `math` element to contain another. These considerations also apply when sub-expressions are passed between applications, such as for cut-and-paste operations; see 7.3 Transferring MathML.

The `math` element can contain an arbitrary number of child elements. They render by default as if they were contained in an `mrow` element.

2.2.1 Attributes

The `math` element accepts any of the attributes that can be set on 3.3.4 Style Change `<mstyle>`, including the common attributes specified in 2.1.6 Attributes Shared by all MathML Elements. In particular, it accepts the `dir` attribute for setting the overall directionality; the `math` element is usually the most useful place to specify the directionality (see 3.1.5 Directionality for further discussion). Note that the `dir` attribute defaults to `ltr` on the `math` element (but *inherits* on all other elements which accept the `dir` attribute); this provides for backward compatibility with MathML 2.0 which had no notion of directionality. Also, it accepts the `mathbackground` attribute in the same sense as `mstyle` and other presentation elements to set the background color of the bounding box, rather than specifying a default for the attribute (see 3.1.9 Mathematics attributes common to presentation elements).

In addition to those attributes, the `math` element accepts:

Name	values	default
displaycore	"block" | "inline"	inline
	specifies whether the enclosed MathML expression should be rendered as a separate vertical block (in display style) or inline, aligned with adjacent text. When `display=block`, `displaystyle` is initialized to `true`, whereas when `display=inline`, `displaystyle` is initialized to `false`; in both cases `scriptlevel` is initialized to 0 (see 3.1.6 Displaystyle and Scriptlevel). Moreover, when the math element is embedded in a larger document, a block math element should be treated as a block element as appropriate for the document type (typically as a new vertical block), whereas an inline math element should be treated as inline (typically exactly as if it were a sequence of words in normal text). In particular, this applies to spacing and linebreaking: for instance, there should not be spaces or line breaks inserted between inline math and any immediately following punctuation. When the display attribute is missing, a rendering agent is free to initialize as appropriate to the context.	
maxwidthnot-core	*length*	*available width*
	specifies the maximum width to be used for linebreaking. The default is the maximum width available in the surrounding environment. If that value cannot be determined, the renderer should assume an infinite rendering width.	
overflownot-core	"linebreak" | "scroll" | "elide" | "truncate" | "scale"	linebreak
	specifies the preferred handing in cases where an expression is too long to fit in the allowed width. See the discussion below.	
altimgnot-core	*URI*	*none*
	provides a URI referring to an image to display as a fall-back for user agents that do not support embedded MathML.	
altimg-widthnot-core	*length*	*width of* `altimg`
	specifies the width to display `altimg`, scaling the image if necessary; see `altimg-height`.	
altimg-heightnot-core	*length*	*height of* `altimg`
	specifies the height to display `altimg`, scaling the image if necessary; if only one of the attributes `altimg-width` and `altimg-height` are given, the scaling should preserve the image's aspect ratio; if neither attribute is given, the image should be shown at its natural size.	
altimg-valignnot-core	*length* | "top" | "middle" | "bottom"	0ex
	specifies the vertical alignment of the image with respect to adjacent inline material. A positive value of `altimg-valign` shifts the bottom of the image above the current baseline, while a negative value lowers it. The keyword "top" aligns the top of the image with the top of adjacent inline material; "center" aligns the middle of the image to the middle of adjacent material; "bottom" aligns the bottom of the image to the bottom of adjacent material (not necessarily the baseline). This attribute only has effect when `display=inline`. By default, the bottom of the image aligns to the baseline.	

alttext[core]	*string*	*none*
	provides a textual alternative as a fall-back for user agents that do not support embedded MathML or images.	
cdgroup[not-core]	*URI*	*none*
	specifies a CD group file that acts as a catalogue of CD bases for locating OpenMath content dictionaries of `csymbol`, `annotation`, and `annotation-xml` elements in this `math` element; see 4.2.3 Content Symbols <csymbol>. When no `cdgroup` attribute is explicitly specified, the document format embedding this `math` element may provide a method for determining CD bases. Otherwise the system must determine a CD base; in the absence of specific information `http://www.openmath.org/cd` is assumed as the CD base for all `csymbol`, `annotation`, and `annotation-xml` elements. This is the CD base for the collection of standard CDs maintained by the OpenMath Society.	

In cases where size negotiation is not possible or fails (for example in the case of an expression that is too long to fit in the allowed width), the `overflow` attribute is provided to suggest a processing method to the renderer. Allowed values are:

Value	Meaning
"linebreak"	The expression will be broken across several lines. See 3.1.7 Linebreaking of Expressions for further discussion. [not-core]
"scroll"	The window provides a viewport into the larger complete display of the mathematical expression. Horizontal or vertical scroll bars are added to the window as necessary to allow the viewport to be moved to a different position. [not-core]
"elide"	The display is abbreviated by removing enough of it so that the remainder fits into the window. For example, a large polynomial might have the first and last terms displayed with “+ ... +” between them. Advanced renderers may provide a facility to zoom in on elided areas. [not-core]
"truncate"	The display is abbreviated by simply truncating it at the right and bottom borders. It is recommended that some indication of truncation is made to the viewer. [not-core]
"scale"	The fonts used to display the mathematical expression are chosen so that the full expression fits in the window. Note that this only happens if the expression is too large. In the case of a window larger than necessary, the expression is shown at its normal size within the larger window. [not-core]

3. Presentation Markup

3.1 Introduction

This chapter specifies the “presentation” elements of MathML, which can be used to describe the layout structure of

mathematical notation.

Most of Presentation Markup is included in [MathML-Core]. That specification should be consulted for the precise details of displaying the elements and attributes that are part of core when displayed in web browsers. Outside of web browsers, MathML presentation elements only suggest (i.e. do not require) specific ways of rendering in order to allow for medium-dependent rendering and for individual preferences of style. Non browser-based renderers are free to use their own layout rules as long as the renderings are intelligible.

The names used for presentation elements are suggestive of their visual layout. However, mathematical notation has a long history of being reused as new concepts are developed. Because of this, an element such as `mfrac` may not actually be a fraction and the `intent` attribute should be used to provide information for auditory renderings.

This chapter describes all of the presentation elements and attributes of MathML along with examples that might clarify usage.

3.1.1 Presentation MathML Structure

The presentation elements are meant to express the syntactic structure of mathematical notation in much the same way as titles, sections, and paragraphs capture the higher-level syntactic structure of a textual document. Because of this, a single row of identifiers and operators will often be represented by multiple nested `mrow` elements rather than a single `mrow`. For example, "$x + a / b$" typically is represented as:

```
<mrow>
  <mi> x </mi>
  <mo> + </mo>
  <mrow>
    <mi> a </mi>
    <mo> / </mo>
    <mi> b </mi>
  </mrow>
</mrow>
```

$$x + a / b$$

Similarly, superscripts are attached to the full expression constituting their base rather than to the just preceding character. This structure permits better-quality rendering of mathematics, especially when details of the rendering environment, such as display widths, are not known ahead of time to the document author. It also greatly eases automatic interpretation of the represented mathematical structures.

Certain characters are used to name identifiers or operators that in traditional notation render the same as other symbols or are rendered invisibly. For example, the characters U+2146, U+2147 and U+2148 represent differential d, exponential e and imaginary i, respectively and are semantically distinct from the same letters used as simple variables. Likewise, the characters U+2061, U+2062, U+2063 and U+2064 represent function application, invisible times, invisible comma and invisible plus . These usually render invisibly but represent significant information that may influence visual spacing and linebreaking, and may have distinct spoken renderings. Accordingly, authors should use these characters (or corresponding entities) wherever applicable.

The complete list of MathML entities is described in [Entities].

3.1.2 Terminology Used In This Chapter

The presentation elements are divided into two classes. *Token elements* represent individual symbols, names, numbers, labels, etc. *Layout schemata* build expressions out of parts and can have only elements as content. These are subdivided into General Layout, Script and Limit, Tabular Math and Elementary Math schemata. There are also a few empty elements used only in conjunction with certain layout schemata.

All individual "symbols" in a mathematical expression should be represented by MathML token elements (e.g., `<mn>24</mn>`). The primary MathML token element types are identifiers (mi, e.g. variables or function names), numbers (mn), and operators (mo, including fences, such as parentheses, and separators, such as commas). There are also token elements used to represent text or whitespace that has more aesthetic than mathematical significance and other elements representing "string literals" for compatibility with computer algebra systems.

The layout schemata specify the way in which sub-expressions are built into larger expressions such as fraction and scripted expressions. Layout schemata attach special meaning to the number and/or positions of their children. A child of a layout schema is also called an *argument* of that element. As a consequence of the above definitions, the content of a layout schema consists exactly of a sequence of zero or more elements that are its arguments.

3.1.3 Required Arguments

Many of the elements described herein require a specific number of arguments (always 1, 2, or 3). In the detailed descriptions of element syntax given below, the number of required arguments is implicitly indicated by giving names for the arguments at various positions. A few elements have additional requirements on the number or type of arguments, which are described with the individual element. For example, some elements accept sequences of zero or more arguments — that is, they are allowed to occur with no arguments at all.

Note that MathML elements encoding rendered space *do* count as arguments of the elements in which they appear. See 3.2.7 Space `<mspace/>` for a discussion of the proper use of such space-like elements.

3.1.3.1 Inferred `<mrow>`s

The elements listed in the following table as requiring 1* argument (`msqrt`, `mstyle`, `merror`, `mpadded`, `mphantom`, `menclose`, `mtd`, `mscarry`, and `math`) conceptually accept a single argument, but actually accept any number of children. If the number of children is 0 or is more than 1, they treat their contents as a single *inferred* `mrow` formed from all their children, and treat this `mrow` as the argument.

For example,

```
<msqrt>
  <mo> − </mo>
  <mn> 1 </mn>
</msqrt>
```

$$\sqrt{-1}$$

is treated as if it were

```
<msqrt>
  <mrow>
    <mo> − </mo>
    <mn> 1 </mn>
  </mrow>
</msqrt>
```

$$\sqrt{-1}$$

This feature allows MathML data not to contain (and its authors to leave out) many `mrow` elements that would otherwise be necessary.

§ *3.1.3.2 Table of argument requirements*

For convenience, here is a table of each element's argument count requirements and the roles of individual arguments when these are distinguished. An argument count of 1* indicates an inferred `mrow` as described above. Although the `math` element is not a presentation element, it is listed below for completeness.

Element	Required argument count	Argument roles (when these differ by position)
mrow	0 or more	
mfrac	2	*numerator denominator*
msqrt	1*	
mroot	2	*base index*
mstyle	1*	
merror	1*	
mpadded	1*	
mphantom	1*	
mfenced	0 or more	
menclose	1*	
msub	2	*base subscript*
msup	2	*base superscript*
msubsup	3	*base subscript superscript*
munder	2	*base underscript*
mover	2	*base overscript*
munderover	3	*base underscript overscript*
mmultiscripts	1 or more	*base* (*subscript superscript*)* [<mprescripts/> (*presubscript presuperscript*)*]
mtable	0 or more rows	0 or more mtr elements
mtr	0 or more	0 or more mtd elements
mtd	1*	
mstack	0 or more	
mlongdiv	3 or more	*divisor result dividend* (*msrow* | *msgroup* | *mscarries* | *msline*)*
msgroup	0 or more	
msrow	0 or more	
mscarries	0 or more	
mscarry	1*	

maction	1 or more	depend on actiontype attribute
math	1*	

3.1.4 Elements with Special Behaviors

Certain MathML presentation elements exhibit special behaviors in certain contexts. Such special behaviors are discussed in the detailed element descriptions below. However, for convenience, some of the most important classes of special behavior are listed here.

Certain elements are considered space-like; these are defined in 3.2.7 Space <mspace/>. This definition affects some of the suggested rendering rules for mo elements (3.2.5 Operator, Fence, Separator or Accent <mo>).

Certain elements, e.g. msup, are able to embellish operators that are their first argument. These elements are listed in 3.2.5 Operator, Fence, Separator or Accent <mo>, which precisely defines an "embellished operator" and explains how this affects the suggested rendering rules for stretchy operators.

3.1.5 Directionality

In the notations familiar to most readers, both the overall layout and the textual symbols are arranged from left to right (LTR). Yet, as alluded to in the introduction, mathematics written in Hebrew or in locales such as Morocco or Persia, the overall layout is used unchanged, but the embedded symbols (often Hebrew or Arabic) are written right to left (RTL). Moreover, in most of the Arabic speaking world, the notation is arranged entirely RTL; thus a superscript is still raised, but it follows the base on the left rather than the right.

MathML 3.0 therefore recognizes two distinct directionalities: the directionality of the text and symbols within token elements and the overall directionality represented by Layout Schemata. These two facets are discussed below.

> NOTE
>
> Probably need to add a little discussion of vertical languages here (and their current lack of support)

3.1.5.1 Overall Directionality of Mathematics Formulas

The overall directionality for a formula, basically the direction of the Layout Schemata, is specified by the dir attribute on the containing math element (see 2.2 The Top-Level <math> Element). The default is ltr. When dir=rtl is used, the layout is simply the mirror image of the conventional European layout. That is, shifts up or down are unchanged, but the progression in laying out is from right to left.

For example, in a RTL layout, sub- and superscripts appear to the left of the base; the surd for a root appears at the right, with the bar continuing over the base to the left. The layout details for elements whose behavior depends on directionality are given in the discussion of the element. In those discussions, the terms leading and trailing are used to specify a side of an

object when which side to use depends on the directionality; i.e. leading means left in LTR but right in RTL. The terms left and right may otherwise be safely assumed to mean left and right.

The overall directionality is usually set on the `math`, but may also be switched for an individual subexpression by using the `dir` attribute on `mrow` or `mstyle` elements. When not specified, all elements inherit the directionality of their container.

3.1.5.2 Bidirectional Layout in Token Elements

The text directionality comes into play for the MathML token elements that can contain text (`mtext`, `mo`, `mi`, `mn` and `ms`) and is determined by the Unicode properties of that text. A token element containing exclusively LTR or RTL characters is displayed straightforwardly in the given direction. When a mixture of directions is involved, such as RTL Arabic and LTR numbers, the Unicode bidirectional algorithm [Bidi] should be applied. This algorithm specifies how runs of characters with the same direction are processed and how the runs are (re)ordered. The base, or initial, direction is given by the overall directionality described above (3.1.5.1 Overall Directionality of Mathematics Formulas) and affects how weakly directional characters are treated and how runs are nested. (The `dir` attribute is thus allowed on token elements to specify the initial directionality that may be needed in rare cases.) Any `mglyph` or `malignmark` elements appearing within a token element are effectively *neutral* and have no effect on ordering.

The important thing to notice is that the bidirectional algorithm is applied independently to the contents of each token element; each token element is an independent run of characters.

Other features of Unicode and scripts that should be respected are ‘mirroring’ and ‘glyph shaping’. Some Unicode characters are marked as being mirrored when presented in a RTL context; that is, the character is drawn as if it were mirrored or replaced by a corresponding character. Thus an opening parenthesis, ‘(’, in RTL will display as ‘)’. Conversely, the solidus (/ U+002F) is *not* marked as mirrored. Thus, an Arabic author that desires the slash to be reversed in an inline division should explicitly use reverse solidus (\ U+005C) or an alternative such as the mirroring DIVISION SLASH (U+2215).

Additionally, calligraphic scripts such as Arabic blend, or connect sequences of characters together, changing their appearance. As this can have a significant impact on readability, as well as aesthetics, it is important to apply such shaping if possible. Glyph shaping, like directionality, applies to each token element's contents individually.

Note that for the transfinite cardinals represented by Hebrew characters, the code points U+2135-U+2138 (ALEF SYMBOL, BET SYMBOL, GIMEL SYMBOL, DALET SYMBOL) should be used in MathML, not the alphabetic look-alike code points. These code points are strong left-to-right.

3.1.6 Displaystyle and Scriptlevel

So-called ‘displayed’ formulas, those appearing on a line by themselves, typically make more generous use of vertical space than inline formulas, which should blend into the adjacent text without intruding into neighboring lines. For example, in a displayed summation, the limits are placed above and below the summation symbol, while when it appears inline the limits would appear in the sub- and superscript position. For similar reasons, sub- and superscripts, nested fractions and other constructs typically display in a smaller size than the main part of the formula. MathML implicitly associates with every presentation node a `displaystyle` and `scriptlevel` reflecting whether a more expansive vertical layout applies and the level of scripting in the current context.

These values are initialized by the `math` element according to the `display` attribute. They are automatically adjusted by the various script and limit schemata elements, and the elements `mfrac` and `mroot`, which typically set `displaystyle` false and increment `scriptlevel` for some or all of their arguments. (See the description for each element for the specific rules used.) They also may be set explicitly via the `displaystyle` and `scriptlevel` attributes on the `mstyle` element or the `displaystyle` attribute of `mtable`. In all other cases, they are inherited from the node's parent.

The `displaystyle` affects the amount of vertical space used to lay out a formula: when true, the more spacious layout of displayed equations is used, whereas when false a more compact layout of inline formula is used. This primarily affects the interpretation of the `largeop` and `movablelimits` attributes of the `mo` element. However, more sophisticated renderers are free to use this attribute to render more or less compactly.

The main effect of `scriptlevel` is to control the font size. Typically, the higher the `scriptlevel`, the smaller the font size. (Non-visual renderers can respond to the font size in an analogous way for their medium.) Whenever the `scriptlevel` is changed, whether automatically or explicitly, the current font size is multiplied by the value of `scriptsizemultiplier` to the power of the *change* in `scriptlevel`. However, changes to the font size due to `scriptlevel` changes should never reduce the size below `scriptminsize` to prevent scripts becoming unreadably small. The default `scriptsizemultiplier` is approximately the square root of 1/2 whereas `scriptminsize` defaults to 8 points; these values may be changed on `mstyle`; see 3.3.4 Style Change <mstyle>. Note that the `scriptlevel` attribute of `mstyle` allows arbitrary values of `scriptlevel` to be obtained, including negative values which result in increased font sizes.

The changes to the font size due to `scriptlevel` should be viewed as being imposed from ‘outside’ the node. This means that the effect of `scriptlevel` is applied before an explicit `mathsize` (see 3.2.2 Mathematics style attributes common to token elements) on a token child of `mfrac`. Thus, the `mathsize` effectively overrides the effect of `scriptlevel`. However, that change to `scriptlevel` changes the current font size, which affects the meaning of an `em` length (see 2.1.5.2 Length Valued Attributes) and so the `scriptlevel` still may have an effect in such cases. Note also that since `mathsize` is not constrained by `scriptminsize`, such direct changes to font size can result in scripts smaller than `scriptminsize`.

Note that direct changes to current font size, whether by CSS or by the `mathsize` attribute (see 3.2.2 Mathematics style attributes common to token elements), have no effect on the value of `scriptlevel`.

TeX's \displaystyle, \textstyle, \scriptstyle, and \scriptscriptstyle correspond to `displaystyle` and `scriptlevel` as `true` and `0`, `false` and `0`, `false` and `1`, and `false` and `2`, respectively. Thus, `math`'s `display=block` corresponds to \displaystyle, while `display=inline` corresponds to \textstyle.

3.1.7 Linebreaking of Expressions

3.1.7.1 Control of Linebreaks

MathML provides support for both automatic and manual (forced) linebreaking of expressions to break excessively long expressions into several lines. All such linebreaks take place within `mrow` (including inferred `mrow`; see 3.1.3.1 Inferred <mrow>s) or `mfenced`. The breaks typically take place at `mo` elements and also, for backwards compatibility, at `mspace`. Renderers may also choose to place automatic linebreaks at other points such as between adjacent `mi` elements or even within a token element such as a very long `mn` element. MathML does not provide a means to specify such linebreaks, but if a renderer chooses to linebreak at such a point, it should indent the following line according to the indentation attributes that

are in effect at that point.

Automatic linebreaking occurs when the containing `math` element has `overflow=linebreak` and the display engine determines that there is not enough space available to display the entire formula. The available width must therefore be known to the renderer. Like font properties, one is assumed to be inherited from the environment in which the MathML element lives. If no width can be determined, an infinite width should be assumed. Inside of an `mtable`, each column has some width. This width may be specified as an attribute or determined by the contents. This width should be used as the line wrapping width for linebreaking, and each entry in an `mtable` is linewrapped as needed.

> ~~ISSUE 304 (CLOSED)~~: Potential presentation MathML items to deprecate in MathML 4
> (mspace's @linebreak)

Forced linebreaks are specified by using `linebreak=newline` on an `mo` or `mspace` element. Both automatic and manual linebreaking can occur within the same formula.

Automatic linebreaking of subexpressions of `mfrac`, `msqrt`, `mroot` and `menclose` and the various script elements is not required. Renderers are free to ignore forced breaks within those elements if they choose.

Attributes on `mo` and possibly on `mspace` elements control linebreaking and indentation of the following line. The aspects of linebreaking that can be controlled are:

- *Where* — attributes determine the desirability of a linebreak at a specific operator or space, in particular whether a break is required or inhibited. These can only be set on `mo` and `mspace` elements. (See 3.2.5.2.2 Linebreaking attributes.)
- *Operator Display/Position* — when a linebreak occurs, determines whether the operator will appear at the end of the line, at the beginning of the next line, or in both positions; and how much vertical space should be added after the linebreak. These attributes can be set on `mo` elements or inherited from `mstyle` or `math` elements. (See 3.2.5.2.2 Linebreaking attributes.)
- *Indentation* — determines the indentation of the line following a linebreak, including indenting so that the next line aligns with some point in a previous line. These attributes can be set on `mo` elements or inherited from `mstyle` or `math` elements. (See 3.2.5.2.3 Indentation attributes.)

When a math element appears in an inline context, it may obey whatever paragraph flow rules are employed by the document's text rendering engine. Such rules are necessarily outside of the scope of this specification. Alternatively, it may use the value of the `math` element's overflow attribute. (See 2.2.1 Attributes.)

§ *3.1.7.2 Examples of Linebreaking*

▸ Show Section

The following example demonstrates forced linebreaks and forced alignment:

```
<mrow>
 <mrow> <mi>f</mi> <mo>&#x2061;<!--ApplyFunction--></mo> <mo>(</mo> <mi>x</mi> <mo>)</mo>
 <mo id='eq1-equals'>=</mo>
 <mrow>
  <msup>
   <mrow> <mo>(</mo> <mrow> <mi>x</mi> <mo>+</mo> <mn>1</mn> </mrow> <mo>)</mo> </mrow>
   <mn>4</mn>
  </msup>
  <mo linebreak='newline' linebreakstyle='before'
      indentalign='id' indenttarget='eq1-equals'>=</mo>
  <mrow>
   <msup> <mi>x</mi> <mn>4</mn> </msup>
   <mo id='eq1-plus'>+</mo>
   <mrow> <mn>4</mn> <mo>&#x2062;<!--InvisibleTimes--></mo> <msup> <mi>x</mi> <mn>3</mn>
   <mo>+</mo>
   <mrow> <mn>6</mn> <mo>&#x2062;<!--InvisibleTimes--></mo> <msup> <mi>x</mi> <mn>2</mn>
   <mo linebreak='newline' linebreakstyle='before'
       indentalignlast='id' indenttarget='eq1-plus'>+</mo>
   <mrow> <mn>4</mn> <mo>&#x2062;<!--InvisibleTimes--></mo> <mi>x</mi> </mrow>
   <mo>+</mo>
   <mn>1</mn>
  </mrow>
 </mrow>
</mrow>
```

This displays as

$$f(x) = (x+1)^4 = x^4 + 4x^3 + 6x^2 + 4x + 1$$

Note that because `indentalignlast` defaults to `indentalign`, in the above example `indentalign` could have been used in place of `indentalignlast`. Also, the specifying `linebreakstyle='before'` is not needed because that is the default value.

3.1.8 Summary of Presentation Elements

3.1.8.1 Token Elements

mi	identifier
mn	number
mo	operator, fence, or separator
mtext	text
mspace	space
ms	string literal

Additionally, the mglyph element may be used within Token elements to represent non-standard symbols as images.

§ *3.1.8.2 General Layout Schemata*

mrow	group any number of sub-expressions horizontally
mfrac	form a fraction from two sub-expressions
msqrt	form a square root (radical without an index)
mroot	form a radical with specified index
mstyle	style change
merror	enclose a syntax error message from a preprocessor
mpadded	adjust space around content
mphantom	make content invisible but preserve its size
mfenced	surround content with a pair of fences
menclose	enclose content with a stretching symbol such as a long division sign

§ *3.1.8.3 Script and Limit Schemata*

msub	attach a subscript to a base
msup	attach a superscript to a base
msubsup	attach a subscript-superscript pair to a base
munder	attach an underscript to a base
mover	attach an overscript to a base
munderover	attach an underscript-overscript pair to a base
mmultiscripts	attach prescripts and tensor indices to a base

§ *3.1.8.4 Tables and Matrices*

mtable	table or matrix
mtr	row in a table or matrix
mtd	one entry in a table or matrix
maligngroup and malignmark	alignment markers

§ *3.1.8.5 Elementary Math Layout*

mstack	columns of aligned characters
mlongdiv	similar to msgroup, with the addition of a divisor and result
msgroup	a group of rows in an mstack that are shifted by similar amounts
msrow	a row in an mstack
mscarries	row in an mstack whose contents represent carries or borrows
mscarry	one entry in an mscarries
msline	horizontal line inside of `mstack`

§ *3.1.8.6 Enlivening Expressions*

`maction`	bind actions to a sub-expression

3.1.9 Mathematics attributes common to presentation elements

In addition to the attributes listed in 2.1.6 Attributes Shared by all MathML Elements, all MathML presentation elements accept the following classes of attribute.

3.1.9.1 MathML Core Attributes

Presentation elements also accept all the Global Attributes specified by [MathML-Core].

These attributes include the following two attributes that are primarily intended for visual media. They are not expected to affect the intended semantics of displayed expressions, but are for use in highlighting or drawing attention to the affected subexpressions. For example, a red "x" is not assumed to be semantically different than a black "x", in contrast to variables with different `mathvariant` values (see 3.2.2 Mathematics style attributes common to token elements).

Name	values	default
mathcolor[core]	*color*	*inherited*
	Specifies the foreground color to use when drawing the components of this element, such as the content for token elements or any lines, surds, or other decorations. It also establishes the default `mathcolor` used for child elements when used on a layout element.	
mathbackground[core]	*color* | "transparent"	transparent
	Specifies the background color to be used to fill in the bounding box of the element and its children. The default, "transparent", lets the background color, if any, used in the current rendering context to show through.	

Since MathML expressions are often embedded in a textual data format such as HTML, the MathML renderer should inherit the foreground color used in the context in which the MathML appears. Note, however, that MathML (in contrast to [MathML-Core]) doesn't specify the mechanism by which style information is inherited from the rendering environment. See 3.2.2 Mathematics style attributes common to token elements for more details.

Note that the suggested MathML visual rendering rules do not define the precise extent of the region whose background is affected by the `mathbackground` attribute, except that, when the content does not have negative dimensions and its drawing region should not overlap with other drawing due to surrounding negative spacing, should lie behind all the drawing done to render the content, and should not lie behind any of the drawing done to render surrounding expressions. The effect of overlap of drawing regions caused by negative spacing on the extent of the region affected by the `mathbackground` attribute is not defined by these rules.

3.2 Token Elements

Token elements in presentation markup are broadly intended to represent the smallest units of mathematical notation which carry meaning. Tokens are roughly analogous to words in text. However, because of the precise, symbolic nature of mathematical notation, the various categories and properties of token elements figure prominently in MathML markup. By contrast, in textual data, individual words rarely need to be marked up or styled specially.

Token elements represent identifiers (`mi`), numbers (`mn`), operators (`mo`), text (`mtext`), strings (`ms`) and spacing (`mspace`). The `mglyph` element may be used *within* token elements to represent non-standard symbols by images. Preceding detailed discussion of the individual elements, the next two subsections discuss the allowable content of token elements and the attributes common to them.

3.2.1 Token Element Content Characters, `<mglyph/>` not-core

Character data in MathML markup is only allowed to occur as part of the content of token elements. Whitespace between elements is ignored. With the exception of the empty `mspace` element, token elements can contain any sequence of zero or more Unicode characters, or `mglyph` or `malignmark` elements. The `mglyph` element is used to represent non-standard characters or symbols by images; the `malignmark` element establishes an alignment point for use within table constructs, and is otherwise invisible (see 3.5.4 Alignment Markers `<maligngroup/>`, `<malignmark/>` not-core).

Characters can be either represented directly as Unicode character data, or indirectly via numeric or character entity references. Unicode contains a number of look-alike characters. See [MathML-Notes] for a discussion of which characters are appropriate to use in which circumstance.

Token elements (other than `mspace`) should be rendered as their content, if any (i.e. in the visual case, as a closely-spaced horizontal row of standard glyphs for the characters or images for the `mglyph`s in their content). An `mspace` element is rendered as a blank space of a width determined by its attributes. Rendering algorithms should also take into account the mathematics style attributes as described below, and modify surrounding spacing by rules or attributes specific to each type of token element. The directional characteristics of the content must also be respected (see 3.1.5.2 Bidirectional Layout in Token Elements).

3.2.1.1 Using images to represent symbols `<mglyph/>` not-core

> NOTE: mglyph is not in MathML-Core
>
> `mglyph` is not supported in [MathML-Core]. In a Web Platform Context it is recommended that the HTML `img` element is used. This is allowed in token elements when MathML is embedded in (X)HTML.
>
> For existing MathML using `mglyph` a Javascript polyfill is provided for Web documents that implements `mglyph` using `img`.

3.2.1.1.1 Description

The `mglyph` element provides a mechanism for displaying images to represent non-standard symbols. It may be used within the content of the token elements `mi`, `mn`, `mo`, `mtext` or `ms` where existing Unicode characters are not adequate.

Unicode defines a large number of characters used in mathematics and, in most cases, glyphs representing these characters are widely available in a variety of fonts. Although these characters should meet almost all users needs, MathML recognizes that mathematics is not static and that new characters and symbols are added when convenient. Characters that become well accepted will likely be eventually incorporated by the Unicode Consortium or other standards bodies, but that is often a lengthy process.

Note that the glyph's `src` attribute uniquely identifies the `mglyph`; two `mglyph`s with the same values for `src` should be considered identical by applications that must determine whether two characters/glyphs are identical.

3.2.1.1.2 Attributes

The `mglyph` element accepts the attributes listed in 3.1.9 Mathematics attributes common to presentation elements, but note that `mathcolor` has no effect. The background color, `mathbackground`, should show through if the specified image has transparency.

`mglyph` also accepts the additional attributes listed here.

Name	values	default
src[not-core]	*URI*	*required*
	Specifies the location of the image resource; it may be a URI relative to the base-URI of the source of the MathML, if any.	
width[not-core]	*length*	*from image*
	Specifies the desired width of the glyph; see `height`.	
height[not-core]	*length*	*from image*
	Specifies the desired height of the glyph. If only one of `width` and `height` are given, the image should be scaled to preserve the aspect ratio; if neither are given, the image should be displayed at its natural size.	
valign[not-core]	*length*	0ex
	Specifies the baseline alignment point of the image with respect to the current baseline. A positive value shifts the bottom of the image above the current baseline while a negative value lowers it. A value of 0 (the default) means that the baseline of the image is at the bottom of the image.	
alt[not-core]	*string*	*required*
	Provides an alternate name for the glyph. If the specified image can't be found or displayed, the renderer may use this name in a warning message or some unknown glyph notation. The name might also be used by an audio renderer or symbol processing system and should be chosen to be descriptive.	

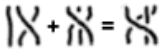
3.2.1.1.3 Example

▸ Show Section

The following example illustrates how a researcher might use the `mglyph` construct with a set of images to work with braid group notation.

```
<mrow>
  <mi><mglyph src="my-braid-23" alt="2 3 braid"/></mi>
  <mo>+</mo>
  <mi><mglyph src="my-braid-132" alt="1 3 2 braid"/></mi>
  <mo>=</mo>
  <mi><mglyph src="my-braid-13" alt="1 3 braid"/></mi>
</mrow>
```

This might render as:



3.2.2 Mathematics style attributes common to token elements

In addition to the attributes defined for all presentation elements (3.1.9 Mathematics attributes common to presentation elements), MathML includes two *mathematics style* attributes as well as a directionality attribute valid on all presentation token elements, as well as the `math` and `mstyle` elements; `dir` is also valid on `mrow` elements. The attributes are:

Name	values	default
mathvariant[core]	"normal" | "bold" | "italic" | "bold-italic" | "double-struck" | "bold-fraktur" | "script" | "bold-script" | "fraktur" | "sans-serif" | "bold-sans-serif" | "sans-serif-italic" | "sans-serif-bold-italic" | "monospace" | "initial" | "tailed" | "looped" | "stretched"	normal (*except on* `<mi>`)
	Specifies the logical class of the token. Note that this class is more than styling, it typically conveys semantic intent; see the discussion below.	
mathsize[core]	"small" | "normal" | "big" | *length*	*inherited*
	Specifies the size to display the token content. The values `small` and `big` choose a size smaller or larger than the current font size, but leave the exact proportions unspecified; `normal` is allowed for completeness, but since it is equivalent to `100%` or `1em`, it has no effect.	
dir[core]	"ltr" | "rtl"	*inherited*
	specifies the initial directionality for text within the token: `ltr` (Left To Right) or `rtl` (Right To Left). This attribute should only be needed in rare cases involving weak or neutral characters; see 3.1.5.1 Overall Directionality of Mathematics Formulas for further discussion. It has no effect on `mspace`.	

The `mathvariant` attribute defines logical classes of token elements. Each class provides a collection of typographically-related symbolic tokens. Each token has a specific meaning within a given mathematical expression and, therefore, needs to be visually distinguished and protected from inadvertent document-wide style changes which might change its meaning. Each token is identified by the combination of the `mathvariant` attribute value and the character data in the token element.

When MathML rendering takes place in an environment where CSS is available, the mathematics style attributes can be viewed as predefined selectors for CSS style rules. See 7.5 Using CSS with MathML for discussion of the interaction of MathML and CSS. Also, see [MathMLforCSS] for discussion of rendering MathML by CSS and a sample CSS style sheet. When CSS is not available, it is up to the internal style mechanism of the rendering application to visually distinguish the different logical classes. Most MathML renderers will probably want to rely on some degree on additional, internal style processing algorithms. In particular, the `mathvariant` attribute does not follow the CSS inheritance model; the default value is `normal` (non-slanted) for all tokens except for `mi` with single-character content. See 3.2.3 Identifier `<mi>` for details.

Renderers have complete freedom in mapping mathematics style attributes to specific rendering properties. However, in practice, the mathematics style attribute names and values suggest obvious typographical properties, and renderers should attempt to respect these natural interpretations as far as possible. For example, it is reasonable to render a token with the `mathvariant` attribute set to `sans-serif` in Helvetica or Arial. However, rendering the token in a Times Roman font could be seriously misleading and should be avoided.

In principle, any `mathvariant` value may be used with any character data to define a specific symbolic token. In practice, only certain combinations of character data and `mathvariant` values will be visually distinguished by a given renderer. For example, there is no clear-cut rendering for a "fraktur alpha" or a "bold italic Kanji" character, and the `mathvariant` values "initial", "tailed", "looped", and "stretched" are appropriate only for Arabic characters.

Certain combinations of character data and `mathvariant` values are equivalent to assigned Unicode code points that encode mathematical alphanumeric symbols. These Unicode code points are the ones in the Arabic Mathematical Alphabetic Symbols block U+1EE00 to U+1EEFF, Mathematical Alphanumeric Symbols block U+1D400 to U+1D7FF, listed in the Unicode standard, and the ones in the Letterlike Symbols range U+2100 to U+214F that represent "holes" in the alphabets in the SMP, listed in 8.2 Mathematical Alphanumeric Symbols. These characters are described in detail in section 2.2 of UTR #25. The description of each such character in the Unicode standard provides an unstyled character to which it would be equivalent except for a font change that corresponds to a `mathvariant` value. A token element that uses the unstyled character in combination with the corresponding `mathvariant` value is equivalent to a token element that uses the mathematical alphanumeric symbol character without the `mathvariant` attribute. Note that the appearance of a mathematical alphanumeric symbol character should not be altered by surrounding `mathvariant` or other style declarations.

Renderers should support those combinations of character data and `mathvariant` values that correspond to Unicode characters, and that they can visually distinguish using available font characters. Renderers may ignore or support those combinations of character data and `mathvariant` values that do not correspond to an assigned Unicode code point, and authors should recognize that support for mathematical symbols that do not correspond to assigned Unicode code points may vary widely from one renderer to another.

Since MathML expressions are often embedded in a textual data format such as HTML, the surrounding text and the MathML must share rendering attributes such as font size, so that the renderings will be compatible in style. For this reason, most attribute values affecting text rendering are inherited from the rendering environment, as shown in the “default” column in the table above. (In cases where the surrounding text and the MathML are being rendered by separate software, e.g. a browser and a plug-in, it is also important for the rendering environment to provide the MathML renderer with additional information, such as the baseline position of surrounding text, which is not specified by any MathML attributes.) Note, however, that MathML doesn't specify the mechanism by which style information is inherited from the rendering environment.

If the requested `mathsize` of the current font is not available, the renderer should approximate it in the manner likely to lead to the most intelligible, highest quality rendering. Note that many MathML elements automatically change the font size in some of their children; see the discussion in 3.1.6 Displaystyle and Scriptlevel.

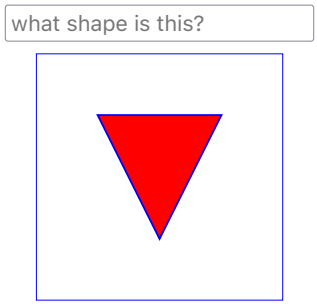
3.2.2.1 Embedding HTML in MathML

MathML can be combined with other formats as described in 7.4 Combining MathML and Other Formats. The recommendation is to embed other formats in MathML by extending the MathML schema to allow additional elements to be children of the `mtext` element or other leaf elements as appropriate to the role they serve in the expression (see 3.2.3

Identifier <mi>, 3.2.4 Number <mn>, and 3.2.5 Operator, Fence, Separator or Accent <mo>). The directionality, font size, and other font attributes should inherit from those that would be used for characters of the containing leaf element (see 3.2.2 Mathematics style attributes common to token elements).

Here is an example of embedding SVG inside of mtext in an HTML context:

```
<mtable>
 <mtr>
  <mtd>
   <mtext><input type="text" placeholder="what shape is this?"/></mtext>
  </mtd>
 </mtr>
 <mtr>
  <mtd>
   <mtext>
    <svg xmlns="http://www.w3.org/2000/svg" width="4cm" height="4cm" viewBox="0 0 400 400
     <rect x="1" y="1" width="398" height="398" style="fill:none; stroke:blue"/>
     <path d="M 100 100 L 300 100 L 200 300 z" style="fill:red; stroke:blue; stroke-width
    </svg>
   </mtext>
  </mtd>
 </mtr>
</mtable>
```



3.2.3 Identifier <mi>core

3.2.3.1 Description

An `mi` element represents a symbolic name or arbitrary text that should be rendered as an identifier. Identifiers can include variables, function names, and symbolic constants. A typical graphical renderer would render an `mi` element as its content (see 3.2.1 Token Element Content Characters, <mglyph/>not-core), with no extra spacing around it (except spacing associated with neighboring elements).

Not all “mathematical identifiers” are represented by `mi` elements — for example, subscripted or primed variables should be represented using `msub` or `msup` respectively. Conversely, arbitrary text playing the role of a “term” (such as an ellipsis in a summed series) should be represented using an `mi` element.

It should be stressed that `mi` is a presentation element, and as such, it only indicates that its content should be rendered as an identifier. In the majority of cases, the contents of an `mi` will actually represent a mathematical identifier such as a variable or function name. However, as the preceding paragraph indicates, the correspondence between notations that should render as identifiers and notations that are actually intended to represent mathematical identifiers is not perfect. For an element whose semantics is guaranteed to be that of an identifier, see the description of `ci` in 4. Content Markup.

3.2.3.2 Attributes

`mi` elements accept the attributes listed in 3.2.2 Mathematics style attributes common to token elements, but in one case with a different default value:

Name	values	default
mathvariant[core]	"normal" | "bold" | "italic" | "bold-italic" | "double-struck" | "bold-fraktur" | "script" | "bold-script" | "fraktur" | "sans-serif" | "bold-sans-serif" | "sans-serif-italic" | "sans-serif-bold-italic" | "monospace" | "initial" | "tailed" | "looped" | "stretched"	*(depends on content; described below)*
	Specifies the logical class of the token. The default is `normal` (non-slanted) unless the content is a single character, in which case it would be `italic`.	

Note that for purposes of determining equivalences of Math Alphanumeric Symbol characters (see 8.2 Mathematical Alphanumeric Symbols) the value of the `mathvariant` attribute should be resolved first, including the special defaulting behavior described above.

3.2.3.3 Examples

▸ Show Section

```
<mi>x</mi>
```

x

```
<mi>D</mi>
```

D

```
<mi>sin</mi>
```

sin

```
<mi mathvariant='script'>L</mi>
```

$\mathcal{L}$

```
<mi></mi>
```

An `mi` element with no content is allowed; `<mi></mi>` might, for example, be used by an "expression editor" to represent a location in a MathML expression which requires a "term" (according to conventional syntax for mathematics) but does not yet contain one.

Identifiers include function names such as "sin". Expressions such as "sin x" should be written using the character U+2061 (entity `af` or `ApplyFunction`) as shown below; see also the discussion of invisible operators in 3.2.5 Operator, Fence, Separator or Accent <mo>.

```
<mrow>
  <mi> sin </mi>
  <mo> &#x2061;<!--ApplyFunction--> </mo>
  <mi> x </mi>
</mrow>
```

sin x

Miscellaneous text that should be treated as a "term" can also be represented by an `mi` element, as in:

```
<mrow>
  <mn> 1 </mn>
  <mo> + </mo>
  <mi> … </mi>
  <mo> + </mo>
  <mi> n </mi>
</mrow>
```

$1 + \ldots + n$

When an `mi` is used in such exceptional situations, explicitly setting the `mathvariant` attribute may give better results than the default behavior of some renderers.

The names of symbolic constants should be represented as `mi` elements:

```
<mi> π </mi>
<mi> 𝑖 </mi>
<mi> 𝑒 </mi>
```

$\pi i e$

3.2.4 Number <mn>core

3.2.4.1 Description

An mn element represents a “numeric literal” or other data that should be rendered as a numeric literal. Generally speaking, a numeric literal is a sequence of digits, perhaps including a decimal point, representing an unsigned integer or real number. A typical graphical renderer would render an mn element as its content (see 3.2.1 Token Element Content Characters, <mglyph/>not-core), with no extra spacing around them (except spacing from neighboring elements such as mo). mn elements are typically rendered in an unslanted font.

The mathematical concept of a “number” can be quite subtle and involved, depending on the context. As a consequence, not all mathematical numbers should be represented using mn; examples of mathematical numbers that should be represented differently are shown below, and include complex numbers, ratios of numbers shown as fractions, and names of numeric constants.

Conversely, since mn is a presentation element, there are a few situations where it may be desirable to include arbitrary text in the content of an mn that should merely render as a numeric literal, even though that content may not be unambiguously interpretable as a number according to any particular standard encoding of numbers as character sequences. As a general rule, however, the mn element should be reserved for situations where its content is actually intended to represent a numeric quantity in some fashion. For an element whose semantics are guaranteed to be that of a particular kind of mathematical number, see the description of cn in 4. Content Markup.

3.2.4.2 Attributes

mn elements accept the attributes listed in 3.2.2 Mathematics style attributes common to token elements.

3.2.4.3 Examples

▸ Show Section

```
<mn> 2 </mn>
```

2

```
<mn> 0.123 </mn>
```

0.123

```
<mn> 1,000,000 </mn>
```

1,000,000

```
<mn> 2.1e10 </mn>
```

2.1e10

```
<mn> 0xFFEF </mn>
```

0xFFEF

```
<mn> MCMLXIX </mn>
```

MCMLXIX

```
<mn> twenty-one </mn>
```

twenty-one

3.2.4.4 Numbers that should not be written using <mn> alone

Many mathematical numbers should be represented using presentation elements other than mn alone; this includes complex numbers, negative numbers, ratios of numbers shown as fractions, and names of numeric constants.

3.2.4.4.1 Examples of complex representations of numbers

▶ Show Section

```
<mrow>
  <mn> 2 </mn>
  <mo> + </mo>
  <mrow>
    <mn> 3 </mn>
    <mo> &#x2062;<!--InvisibleTimes--> </mo>
    <mi> 𝑖 </mi>
  </mrow>
</mrow>
```

$$2 + 3i$$

```
<mfrac> <mn> 1 </mn> <mn> 2 </mn> </mfrac>
```

$$\frac{1}{2}$$

```
<mrow><mo>-</mo><mn>2</mn></mrow>
```

$$-2$$

```
<mi> π </mi>
```

$$\pi$$

```
<mi> 𝑒 </mi>
```

$$e$$

3.2.5 Operator, Fence, Separator or Accent <mo>core

3.2.5.1 Description

An `mo` element represents an operator or anything that should be rendered as an operator. In general, the notational conventions for mathematical operators are quite complicated, and therefore MathML provides a relatively sophisticated mechanism for specifying the rendering behavior of an `mo` element. As a consequence, in MathML the list of things that should "render as an operator" includes a number of notations that are not mathematical operators in the ordinary sense. Besides ordinary operators with infix, prefix, or postfix forms, these include fence characters such as braces, parentheses, and "absolute value" bars; separators such as comma and semicolon; and mathematical accents such as a bar or tilde over a symbol. We will use the term "operator" in this chapter to refer to operators in this broad sense.

Typical graphical renderers show all `mo` elements as the content (see 3.2.1 Token Element Content Characters, `<mglyph/>`[not-core]), with additional spacing around the element determined by its attributes and further described below. Renderers without access to complete fonts for the MathML character set may choose to render an `mo` element as not precisely the characters in its content in some cases. For example, `<mo> ≤ </mo>` might be rendered as <= to a terminal. However, as a general rule, renderers should attempt to render the content of an `mo` element as literally as possible. That is, `<mo> ≤ </mo>` and `<mo> &lt;= </mo>` should render differently. The first one should render as a single character representing a less-than-or-equal-to sign, and the second one as the two-character sequence <=.

A key feature of the `mo` element is that its default attribute values are set on a case-by-case basis from an "operator dictionary" as explained below. In particular, default value for `stretch`, `symmetric` and `accent` can usually be found in the operator dictionary and therefore need not be specified on each `mo` element.

Note that some mathematical operators are represented not by `mo` elements alone, but by `mo` elements "embellished" with (for example) surrounding superscripts; this is further described below. Conversely, as presentation elements, `mo` elements can contain arbitrary text, even when that text has no standard interpretation as an operator; for an example, see the discussion "Mixing text and mathematics" in 3.2.6 Text `<mtext>`. See also 4. Content Markup for definitions of MathML content elements that are guaranteed to have the semantics of specific mathematical operators.

Note also that linebreaking, as discussed in 3.1.7 Linebreaking of Expressions, usually takes place at operators (either before or after, depending on local conventions). Thus, `mo` accepts attributes to encode the desirability of breaking at a particular operator, as well as attributes describing the treatment of the operator and indentation in case a linebreak is made at that operator.

3.2.5.2 Attributes

`mo` elements accept the attributes listed in 3.2.2 Mathematics style attributes common to token elements and the additional attributes listed here. Since the display of operators is so critical in mathematics, the `mo` element accepts a large number of attributes; these are described in the next three subsections.

Most attributes get their default values from an enclosing `mstyle` element, `math` element, from the containing document, or from 3.2.5.6.1 The operator dictionary. When a value that is listed as "inherited" is not explicitly given on an `mo`, `mstyle` element, `math` element, or found in the operator dictionary for a given `mo` element, the default value shown in parentheses is used.

§ 3.2.5.2.1 DICTIONARY-BASED ATTRIBUTES

Name	values	default
form[core]	"prefix" | "infix" | "postfix"	*set by position of operator in an* `mrow`
	Specifies the role of the operator in the enclosing expression. This role and the operator content affect the lookup of the operator in the operator dictionary which affects the spacing and other default properties; see 3.2.5.6.2 Default value of the `form` attribute.	
lspace[core]	*length*	*set by dictionary* (thickmathspace)
	Specifies the leading space appearing before the operator; see 3.2.5.6.4 Spacing around an operator. (Note that before is on the right in a RTL context; see 3.1.5 Directionality.)	
rspace[core]	*length*	*set by dictionary* (thickmathspace)
	Specifies the trailing space appearing after the operator; see 3.2.5.6.4 Spacing around an operator. (Note that after is on the left in a RTL context; see 3.1.5 Directionality.)	
stretchy[core]	"true" | "false"	*set by dictionary* (false)
	Specifies whether the operator should stretch to the size of adjacent material; see 3.2.5.7 Stretching of operators, fences and accents.	
symmetric[core]	"true" | "false"	*set by dictionary* (false)
	Specifies whether the operator should be kept symmetric around the math axis when stretchy. Note this property only applies to vertically stretched symbols. See 3.2.5.7 Stretching of operators, fences and accents.	
maxsize[core]	*length*	*set by dictionary* (unbounded)
	Specifies the maximum size of the operator when stretchy; see 3.2.5.7 Stretching of operators, fences and accents. If not given, the maximum size is unbounded. Unitless or percentage values indicate a multiple of the reference size, being the size of the unstretched glyph. MathML 4 deprecates "infinity" as possible value as it is the same as not providing a value.	
minsize[core]	*length*	*set by dictionary* (100%)
	Specifies the minimum size of the operator when stretchy; see 3.2.5.7 Stretching of operators, fences and accents. Unitless or percentage values indicate a multiple of the reference size, being the size of the unstretched glyph.	
largeop[core]	"true" | "false"	*set by dictionary* (false)
	Specifies whether the operator is considered a 'large' operator, that is, whether it should be drawn larger than normal when `displaystyle=true` (similar to using TeX's `\displaystyle`). Examples of large operators include U+222B and U+220F (entities `int` and `prod`). See 3.1.6 Displaystyle and Scriptlevel for more discussion.	
movablelimits[core]	"true" | "false"	*set by dictionary* (false)
	Specifies whether under- and overscripts attached to this operator 'move' to the more compact sub- and superscript positions when `displaystyle` is false. Examples of operators that typically	

	sub- and superscript positions when `displaystyle` is false. Examples of operators that typically have `movablelimits=true` are U+2211 and U+220F (entitites `sum`, `prod`), as well as `lim`. See 3.1.6 Displaystyle and Scriptlevel for more discussion.	
accent[not-core]	"true" | "false"	*set by dictionary* (false)
	Specifies whether this operator should be treated as an accent (diacritical mark) when used as an underscript or overscript; see `munder`, `mover` and `munderover`. Note: for compatibility with MathML Core, use `accent=true` on the enclosing `mover` and `munderover` in place of this attribute.	

3.2.5.2.2 Linebreaking attributes

The following attributes affect when a linebreak does or does not occur, and the appearance of the linebreak when it does occur.

Name	values	default
linebreak[not-core]	"auto" | "newline" | "nobreak" | "goodbreak" | "badbreak"	auto
	Specifies the desirability of a linebreak occurring at this operator: the default `auto` indicates the renderer should use its default linebreaking algorithm to determine whether to break; `newline` is used to force a linebreak; for automatic linebreaking, `nobreak` forbids a break; `goodbreak` suggests a good position; `badbreak` suggests a poor position.	
lineleading[not-core]	*length*	*inherited* (100%)
	Specifies the amount of vertical space to use after a linebreak. For tall lines, it is often clearer to use more leading at linebreaks. Rendering agents are free to choose an appropriate default.	
linebreakstyle[not-core]	"before" | "after" | "duplicate" | "infixlinebreakstyle"	*set by dictionary* (before)
	Specifies whether a linebreak occurs 'before' or 'after' the operator when a linebreak occurs on this operator; or whether the operator is duplicated. `before` causes the operator to appear at the beginning of the new line (but possibly indented); `after` causes it to appear at the end of the line before the break. `duplicate` places the operator at both positions. `infixlinebreakstyle` uses the value that has been specified for infix operators; this value (one of `before`, `after` or `duplicate`) can be specified by the application or bound by `mstyle` (`before` corresponds to the most common style of linebreaking).	
linebreakmultchar[not-core]	*string*	*inherited* (⁢)
	Specifies the character used to make an ⁢ operator visible at a linebreak. For example, `linebreakmultchar="·"` would make the multiplication visible as a center dot.	

`linebreak` values on adjacent `mo` and `mspace` elements do not interact; `linebreak=nobreak` on an `mo` does not, in itself, inhibit a break on a preceding or following (possibly nested) `mo` or `mspace` element and does not interact with the `linebreakstyle` attribute value of the preceding or following `mo` element. It does prevent breaks from occurring on either side of the `mo` element in all other situations.

§ 3.2.5.2.3 Indentation attributes

The following attributes affect indentation of the lines making up a formula. Primarily these attributes control the positioning of new lines following a linebreak, whether automatic or manual. However, `indentalignfirst` and `indentshiftfirst` also control the positioning of a single line formula without any linebreaks. When these attributes appear on `mo` or `mspace` they apply if a linebreak occurs at that element. When they appear on `mstyle` or `math` elements, they determine defaults for the style to be used for any linebreaks occurring within. Note that except for cases where heavily

marked-up manual linebreaking is desired, many of these attributes are most useful when bound on an `mstyle` or `math` element.

Note that since the rendering context, such as the available width and current font, is not always available to the author of the MathML, a renderer may ignore the values of these attributes if they result in a line in which the remaining width is too small to usefully display the expression or if they result in a line in which the remaining width exceeds the available linewrapping width.

Name	values	default
indentalign[not-core]	"left" | "center" | "right" | "auto" | "id"	*inherited* (auto)
	Specifies the positioning of lines when linebreaking takes place within an `mrow`; see below for discussion of the attribute values.	
indentshift[not-core]	*length*	*inherited* (0)
	Specifies an additional indentation offset relative to the position determined by `indentalign`. When the value is a percentage value, the value is relative to the horizontal space that a MathML renderer has available, this is the current target width as used for linebreaking as specified in 3.1.7 Linebreaking of Expressions. Note: numbers without units were allowed in MathML 3 and treated similarly to percentage values, but unitless numbers are deprecated in MathML 4.	
indenttarget[not-core]	*idref*	*inherited (none)*
	Specifies the id of another element whose horizontal position determines the position of indented lines when `indentalign=id`. Note that the identified element may be outside of the current `math` element, allowing for inter-expression alignment, or may be within invisible content such as `mphantom`; it must appear *before* being referenced, however. This may lead to an id being unavailable to a given renderer or in a position that does not allow for alignment. In such cases, the `indentalign` should revert to `auto`.	
indentalignfirst[not-core]	"left" | "center" | "right" | "auto" | "id" | "indentalign"	*inherited* (indentalign)
	Specifies the indentation style to use for the first line of a formula; the value `indentalign` (the default) means to indent the same way as used for the general line.	
indentshiftfirst[not-core]	*length* | "indentshift"	*inherited* (indentshift)
	Specifies the offset to use for the first line of a formula; the value `indentshift` (the default) means to use the same offset as used for the general line. Percentage values and numbers without unit are interpreted as described for `indentshift`.	
indentalignlast[not-core]	"left" | "center" | "right" | "auto" | "id" | "indentalign"	*inherited* (indentalign)
	Specifies the indentation style to use for the last line when a linebreak occurs within a given `mrow`; the value `indentalign` (the default) means to indent the same way as used for the general line. When there are exactly two lines, the value of this attribute should be used for the second line in preference to `indentalign`.	
indentshiftlast[not-core]	*length* | "indentshift"	*inherited* (indentshift)
	Specifies the offset to use for the last line when a linebreak occurs within a given `mrow`; the value `indentshift` (the default) means to indent the same way as used for the general line. When there are exactly two lines, the value of this attribute should be used for the second line in preference to `indentshift`. Percentage values and numbers without unit are interpreted as described for `indentshift`.	

The legal values of indentalign are:

Value	Meaning
left	Align the left side of the next line to the left side of the line wrapping width
center	Align the center of the next line to the center of the line wrapping width
right	Align the right side of the next line to the right side of the line wrapping width
auto	(default) indent using the renderer's default indenting style; this may be a fixed amount or one that varies with the depth of the element in the mrow nesting or some other similar method.
id	Align the left side of the next line to the left side of the element referenced by the *idref* (given by `indenttarget`); if no such element exists, use `auto` as the `indentalign` value

3.2.5.3 Examples with ordinary operators

▶ Show Section

```
<mo> + </mo>
```

+

```
<mo> &lt; </mo>
```

<

```
<mo> ≤ </mo>
```

≤

```
<mo> &lt;= </mo>
```

<=

```
<mo> ++ </mo>
```

++

```
<mo> ∑ </mo>
```

∑

```
<mo> .NOT. </mo>
```

.NOT.

```
<mo> and </mo>
```

and

```
<mo> &#x2062;<!--InvisibleTimes--> </mo>
```

```
<mo mathvariant='bold'> + </mo>
```

+

3.2.5.4 Examples with fences and separators

▸ Show Section

Note that the `mo` elements in these examples don't need explicit `stretchy` or `symmetric` attributes, since these can be found using the operator dictionary as described below. Some of these examples could also be encoded using the `mfenced` element described in 3.3.8 Expression Inside Pair of Fences <mfenced>[not-core].

(*a*+*b*)

```
<mrow>
  <mo> ( </mo>
  <mrow>
    <mi> a </mi>
    <mo> + </mo>
    <mi> b </mi>
  </mrow>
  <mo> ) </mo>
</mrow>
```

$$(a + b)$$

[0,1)

```
<mrow>
  <mo> [ </mo>
  <mrow>
    <mn> 0 </mn>
    <mo> , </mo>
    <mn> 1 </mn>
  </mrow>
  <mo> ) </mo>
</mrow>
```

$$[0, 1)$$

f(*x*,*y*)

```
<mrow>
  <mi> f </mi>
  <mo> &#x2061;<!--ApplyFunction--> </mo>
  <mrow>
    <mo> ( </mo>
    <mrow>
      <mi> x </mi>
      <mo> , </mo>
      <mi> y </mi>
    </mrow>
    <mo> ) </mo>
  </mrow>
</mrow>
```

$$f(x, y)$$

3.2.5.5 Invisible operators

§

Certain operators that are “invisible” in traditional mathematical notation should be represented using specific characters (or entity references) within `mo` elements, rather than simply by nothing. The characters used for these “invisible operators” are:

Character	Entity name	Short name
U+2061	`ApplyFunction`	`af`
U+2062	`InvisibleTimes`	`it`
U+2063	`InvisibleComma`	`ic`
U+2064		

§ 3.2.5.5.1 Examples

▸ Show Section

The MathML representations of the examples in the above table are:

```
<mrow>
  <mi> f </mi>
  <mo> &#x2061;<!--ApplyFunction--> </mo>
  <mrow>
    <mo> ( </mo>
    <mi> x </mi>
    <mo> ) </mo>
  </mrow>
</mrow>
```

$$f(x)$$

```
<mrow>
  <mi> sin </mi>
  <mo> &#x2061;<!--ApplyFunction--> </mo>
  <mi> x </mi>
</mrow>
```

$$\sin x$$

```
<mrow>
  <mi> x </mi>
  <mo> &#x2062;<!--InvisibleTimes--> </mo>
  <mi> y </mi>
</mrow>
```

$$xy$$

```
<msub>
  <mi> m </mi>
  <mrow>
    <mn> 1 </mn>
    <mo> &#x2063;<!--InvisibleComma--> </mo>
    <mn> 2 </mn>
  </mrow>
</msub>
```

$$m_{12}$$

```
<mrow>
  <mn> 2 </mn>
  <mo> &#x2064; </mo>
  <mfrac>
    <mn> 3 </mn>
    <mn> 4 </mn>
  </mfrac>
</mrow>
```

$$2\frac{3}{4}$$

3.2.5.6 Detailed rendering rules for <mo> elements

Typical visual rendering behaviors for mo elements are more complex than for the other MathML token elements, so the rules for rendering them are described in this separate subsection.

Note that, like all rendering rules in MathML, these rules are suggestions rather than requirements. The description below is given to make the intended effect of the various rendering attributes as clear as possible. Detailed layout rules for browser implementations for operators are given in MathML Core.

3.2.5.6.1 The operator dictionary

Many mathematical symbols, such as an integral sign, a plus sign, or a parenthesis, have a well-established, predictable, traditional notational usage. Typically, this usage amounts to certain default attribute values for `mo` elements with specific contents and a specific `form` attribute. Since these defaults vary from symbol to symbol, MathML anticipates that renderers will have an "operator dictionary" of default attributes for `mo` elements (see B. Operator Dictionary) indexed by each `mo` element's content and `form` attribute. If an `mo` element is not listed in the dictionary, the default values shown in parentheses in the table of attributes for `mo` should be used, since these values are typically acceptable for a generic operator.

Some operators are "overloaded", in the sense that they can occur in more than one form (prefix, infix, or postfix), with possibly different rendering properties for each form. For example, "+" can be either a prefix or an infix operator. Typically, a visual renderer would add space around both sides of an infix operator, while only in front of a prefix operator. The `form` attribute allows specification of which form to use, in case more than one form is possible according to the operator dictionary and the default value described below is not suitable.

3.2.5.6.2 Default value of the `form` attribute

The `form` attribute does not usually have to be specified explicitly, since there are effective heuristic rules for inferring the value of the `form` attribute from the context. If it is not specified, and there is more than one possible form in the dictionary for an `mo` element with given content, the renderer should choose which form to use as follows (but see the exception for embellished operators, described later):

- If the operator is the first argument in an `mrow` with more than one argument (ignoring all space-like arguments (see 3.2.7 Space <mspace/>) in the determination of both the length and the first argument), the prefix form is used;
- if it is the last argument in an `mrow` with more than one argument (ignoring all space-like arguments), the postfix form is used;
- if it is the only element in an implicit or explicit `mrow` and if it is in a script position of one of the elements listed in 3.4 Script and Limit Schemata, the postfix form is used;
- in all other cases, including when the operator is not part of an `mrow`, the infix form is used.

Note that the `mrow` discussed above may be *inferred*; see 3.1.3.1 Inferred <mrow>s.

Opening fences should have `form="prefix"`, and closing fences should have `form="postfix"`; separators are usually "infix", but not always, depending on their surroundings. As with ordinary operators, these values do not usually need to be specified explicitly.

If the operator does not occur in the dictionary with the specified form, the renderer should use one of the forms that is available there, in the order of preference: infix, postfix, prefix; if no forms are available for the given `mo` element content, the renderer should use the defaults given in parentheses in the table of attributes for `mo`.

3.2.5.6.3 Exception for embellished operators

There is one exception to the above rules for choosing an `mo` element's default `form` attribute. An `mo` element that is "embellished" by one or more nested subscripts, superscripts, surrounding text or whitespace, or style changes behaves differently. It is the embellished operator as a whole (this is defined precisely, below) whose position in an `mrow` is examined by the above rules and whose surrounding spacing is affected by its form, not the `mo` element at its core; however, the attributes influencing this surrounding spacing are taken from the `mo` element at the core (or from that element's dictionary entry).

For example, the "$+_4$" in $a +_4 b$ should be considered an infix operator as a whole, due to its position in the middle of an `mrow`, but its rendering attributes should be taken from the `mo` element representing the "+", or when those are not specified explicitly, from the operator dictionary entry for `<mo form="infix"> + </mo>`. The precise definition of an "***embellished operator***" is:

- an `mo` element;
- or one of the elements `msub`, `msup`, `msubsup`, `munder`, `mover`, `munderover`, `mmultiscripts`, `mfrac`, or `semantics` (6.5 The <semantics> element), whose first argument exists and is an embellished operator;
- or one of the elements `mstyle`, `mphantom`, or `mpadded`, such that an `mrow` containing the same arguments would be an embellished operator;
- or an `maction` element whose selected sub-expression exists and is an embellished operator;
- or an `mrow` whose arguments consist (in any order) of one embellished operator and zero or more space-like elements.

Note that this definition permits nested embellishment only when there are no intervening enclosing elements not in the above list.

The above rules for choosing operator forms and defining embellished operators are chosen so that in all ordinary cases it will not be necessary for the author to specify a `form` attribute.

3.2.5.6.4 Spacing around an operator

The amount of horizontal space added around an operator (or embellished operator), when it occurs in an `mrow`, can be directly specified by the `lspace` and `rspace` attributes. Note that `lspace` and `rspace` should be interpreted as leading and trailing space, in the case of RTL direction. By convention, operators that tend to bind tightly to their arguments have smaller values for spacing than operators that tend to bind less tightly. This convention should be followed in the operator dictionary included with a MathML renderer.

Some renderers may choose to use no space around most operators appearing within subscripts or superscripts, as is done in TeX.

Non-graphical renderers should treat spacing attributes, and other rendering attributes described here, in analogous ways for their rendering medium. For example, more space might translate into a longer pause in an audio rendering.

3.2.5.7 Stretching of operators, fences and accents

Four attributes govern whether and how an operator (perhaps embellished) stretches so that it matches the size of other elements: `stretchy`, `symmetric`, `maxsize`, and `minsize`. If an operator has the attribute `stretchy=true`, then it (that is, each character in its content) obeys the stretching rules listed below, given the constraints imposed by the fonts and font rendering system. In practice, typical renderers will only be able to stretch a small set of characters, and quite possibly will only be able to generate a discrete set of character sizes.

There is no provision in MathML for specifying in which direction (horizontal or vertical) to stretch a specific character or operator; rather, when `stretchy=true` it should be stretched in each direction for which stretching is possible and reasonable for that character. It is up to the renderer to know in which directions it is reasonable to stretch a character, if it can stretch the character. Most characters can be stretched in at most one direction by typical renderers, but some renderers may be able to stretch certain characters, such as diagonal arrows, in both directions independently.

The `minsize` and `maxsize` attributes limit the amount of stretching (in either direction). These two attributes are given as multipliers of the operator's normal size in the direction or directions of stretching, or as absolute sizes using units. For example, if a character has `maxsize=300%`, then it can grow to be no more than three times its normal (unstretched) size.

The `symmetric` attribute governs whether the height and depth above and below the axis of the character are forced to be equal (by forcing both height and depth to become the maximum of the two). An example of a situation where one might set `symmetric=false` arises with parentheses around a matrix not aligned on the axis, which frequently occurs when multiplying non-square matrices. In this case, one wants the parentheses to stretch to cover the matrix, whereas stretching the parentheses symmetrically would cause them to protrude beyond one edge of the matrix. The `symmetric` attribute only applies to characters that stretch vertically (otherwise it is ignored).

If a stretchy `mo` element is embellished (as defined earlier in this section), the `mo` element at its core is stretched to a size based on the context of the embellished operator as a whole, i.e. to the same size as if the embellishments were not present. For example, the parentheses in the following example (which would typically be set to be stretchy by the operator dictionary) will be stretched to the same size as each other, and the same size they would have if they were not underlined and overlined, and furthermore will cover the same vertical interval:

```
<mrow>
  <munder>
    <mo> ( </mo>
    <mo> _ </mo>
  </munder>
  <mfrac>
    <mi> a </mi>
    <mi> b </mi>
  </mfrac>
  <mover>
    <mo> ) </mo>
    <mo> ¯ </mo>
  </mover>
</mrow>
```

$$\underline{\left(\frac{a}{b}\right)}^{\overline{}}$$

Note that this means that the stretching rules given below must refer to the context of the embellished operator as a whole, not just to the `mo` element itself.

3.2.5.7.1 Example of stretchy attributes

▸ Show Section

This shows one way to set the maximum size of a parenthesis so that it does not grow, even though its default value is `stretchy=true`.

```
<mrow>
  <mo maxsize="100%">(</mo>
  <mfrac>
    <msup><mi>a</mi><mn>2</mn></msup>
    <msup><mi>b</mi><mn>2</mn></msup>
  </mfrac>
  <mo maxsize="100%">)</mo>
</mrow>
```

$$\left(\frac{a^2}{b^2}\right)$$

The above should render as $\left(\frac{a^2}{b^2}\right)$ as opposed to the default rendering $\left(\frac{a^2}{b^2}\right)$.

Note that each parenthesis is sized independently; if only one of them had `maxsize=100%`, they would render with different sizes.

3.2.5.7.2 Vertical Stretching Rules

The general rules governing stretchy operators are:

- If a stretchy operator is a direct sub-expression of an `mrow` element, or is the sole direct sub-expression of an `mtd` element in some row of a table, then it should stretch to cover the height and depth (above and below the axis) of the *non*-stretchy direct sub-expressions in the `mrow` element or table row, unless stretching is constrained by `minsize` or `maxsize` attributes.
- In the case of an embellished stretchy operator, the preceding rule applies to the stretchy operator at its core.
- The preceding rules also apply in situations where the `mrow` element is inferred.
- The rules for symmetric stretching only apply if `symmetric=true` and if the stretching occurs in an `mrow` or in an `mtr` whose `rowalign` value is either `baseline` or `axis`.

The following algorithm specifies the height and depth of vertically stretched characters:

1. Let `maxheight` and `maxdepth` be the maximum height and depth of the *non*-stretchy siblings within the same `mrow` or `mtr`. Let axis be the height of the math axis above the baseline.

   Note that even if a `minsize` or `maxsize` value is set on a stretchy operator, it is *not* used in the initial calculation of the maximum height and depth of an `mrow`.

2. If `symmetric=true`, then the computed height and depth of the stretchy operator are:

   ```
   height=max(maxheight−axis, maxdepth+axis) + axis
   depth =max(maxheight−axis, maxdepth+axis) − axis
   ```

   Otherwise the height and depth are:

   ```
   height= maxheight
   depth = maxdepth
   ```

3. If the total size = height+depth is less than minsize or greater than maxsize, increase or decrease both height and depth proportionately so that the effective size meets the constraint.

By default, most vertical arrows, along with most opening and closing fences are defined in the operator dictionary to stretch by default.

In the case of a stretchy operator in a table cell (i.e. within an `mtd` element), the above rules assume each cell of the table row containing the stretchy operator covers exactly one row. (Equivalently, the value of the `rowspan` attribute is assumed to be 1 for all the table cells in the table row, including the cell containing the operator.) When this is not the case, the operator should only be stretched vertically to cover those table cells that are entirely within the set of table rows that the operator's cell covers. Table cells that extend into rows not covered by the stretchy operator's table cell should be ignored. See 3.5.3.2 Attributes for details about the `rowspan` attribute.

3.2.5.7.3 Horizontal Stretching Rules

- If a stretchy operator, or an embellished stretchy operator, is a direct sub-expression of an `munder`, `mover`, or `munderover` element, or if it is the sole direct sub-expression of an `mtd` element in some column of a table (see `mtable`), then it, or the `mo` element at its core, should stretch to cover the width of the other direct sub-expressions in the given element (or in the same table column), given the constraints mentioned above.
- In the case of an embellished stretchy operator, the preceding rule applies to the stretchy operator at its core.

By default, most horizontal arrows and some accents stretch horizontally.

In the case of a stretchy operator in a table cell (i.e. within an `mtd` element), the above rules assume each cell of the table column containing the stretchy operator covers exactly one column. (Equivalently, the value of the `columnspan` attribute is assumed to be 1 for all the table cells in the table row, including the cell containing the operator.) When this is not the case, the operator should only be stretched horizontally to cover those table cells that are entirely within the set of table columns that the operator's cell covers. Table cells that extend into columns not covered by the stretchy operator's table cell should be

ignored. See 3.5.3.2 Attributes for details about the `rowspan` attribute.

The rules for horizontal stretching include `mtd` elements to allow arrows to stretch for use in commutative diagrams laid out using `mtable`. The rules for the horizontal stretchiness include scripts to make examples such as the following work:

```
<mrow>
  <mi> x </mi>
  <munder>
    <mo> → </mo>
    <mtext> maps to </mtext>
  </munder>
  <mi> y </mi>
</mrow>
```

$$x \underset{\text{maps to}}{\longrightarrow} y$$

3.2.5.7.4 Rules Common to both Vertical and Horizontal Stretching

If a stretchy operator is not required to stretch (i.e. if it is not in one of the locations mentioned above, or if there are no other expressions whose size it should stretch to match), then it has the standard (unstretched) size determined by the font and current `mathsize`.

If a stretchy operator is required to stretch, but all other expressions in the containing element (as described above) are also stretchy, all elements that can stretch should grow to the maximum of the normal unstretched sizes of all elements in the containing object, if they can grow that large. If the value of `minsize` or `maxsize` prevents that, then the specified (min or max) size is used.

For example, in an `mrow` containing nothing but vertically stretchy operators, each of the operators should stretch to the maximum of all of their normal unstretched sizes, provided no other attributes are set that override this behavior. Of course, limitations in fonts or font rendering may result in the final, stretched sizes being only approximately the same.

3.2.6 Text `<mtext>`[core]

3.2.6.1 Description

An `mtext` element is used to represent arbitrary text that should be rendered as itself. In general, the `mtext` element is intended to denote commentary text.

Note that text with a clearly defined notational role might be more appropriately marked up using mi or mo.

An `mtext` element can also contain “renderable whitespace”, i.e. invisible characters that are intended to alter the positioning of surrounding elements. In non-graphical media, such characters are intended to have an analogous effect, such

as introducing positive or negative time delays or affecting rhythm in an audio renderer. However, see 2.1.7 Collapsing Whitespace in Input.

3.2.6.2 Attributes

`mtext` elements accept the attributes listed in 3.2.2 Mathematics style attributes common to token elements.

See also the warnings about the legal grouping of "space-like elements" in 3.2.7 Space `<mspace/>`, and about the use of such elements for "tweaking" in [MathML-Notes].

3.2.6.3 Examples

▸ Show Section

```
<mrow>
  <mtext> Theorem 1: </mtext>
  <mtext>  <!--ThinSpace--> </mtext>
  <mtext>  <!--ThickSpace--> <!--ThickSpace--> </mtext>
  <mtext> /* a comment */ </mtext>
</mrow>
```

Theorem 1: /* a comment */

3.2.7 Space `<mspace/>` core

3.2.7.1 Description

An `mspace` empty element represents a blank space of any desired size, as set by its attributes. It can also be used to make linebreaking suggestions to a visual renderer. Note that the default values for attributes have been chosen so that they typically will have no effect on rendering. Thus, the `mspace` element is generally used with one or more attribute values explicitly specified.

Note the warning about the legal grouping of "space-like elements" given below, and the warning about the use of such elements for "tweaking" in [MathML-Notes]. See also the other elements that can render as whitespace, namely `mtext`, `mphantom`, and `maligngroup`.

3.2.7.2 Attributes

In addition to the attributes listed below, `mspace` elements accept the attributes described in 3.2.2 Mathematics style attributes common to token elements, but note that `mathvariant` and `mathcolor` have no effect and that `mathsize` only affects the interpretation of units in sizing attributes (see 2.1.5.2 Length Valued Attributes). `mspace` also accepts the indentation attributes described in 3.2.5.2.3 Indentation attributes.

Name	values	default
width[core]	*length*	0em
	Specifies the desired width of the space.	
height[core]	*length*	0ex
	Specifies the desired height (above the baseline) of the space.	
depth[core]	*length*	0ex
	Specifies the desired depth (below the baseline) of the space.	

Linebreaking was originally specified on `mspace` in MathML2, but much greater control over linebreaking and indentation was add to `mo` in MathML 3. Linebreaking on `mspace` is deprecated in MathML 4.

§ *3.2.7.3 Examples*

▸ Show Section

```
<mspace height="3ex" depth="2ex"/>
```

§ *3.2.7.4 Definition of space-like elements*

A number of MathML presentation elements are “space-like” in the sense that they typically render as whitespace, and do not affect the mathematical meaning of the expressions in which they appear. As a consequence, these elements often function in somewhat exceptional ways in other MathML expressions. For example, space-like elements are handled specially in the suggested rendering rules for `mo` given in 3.2.5 Operator, Fence, Separator or Accent <mo>. The following MathML elements are defined to be “space-like”:

- an `mtext`, `mspace`, `maligngroup`, or `malignmark` element;
- an `mstyle`, `mphantom`, or `mpadded` element, all of whose direct sub-expressions are space-like;
- a `semantics` element whose first argument exists and is space-like;
- an `maction` element whose selected sub-expression exists and is space-like;
- an `mrow` all of whose direct sub-expressions are space-like.

Note that an `mphantom` is *not* automatically defined to be space-like, unless its content is space-like. This is because operator spacing is affected by whether adjacent elements are space-like. Since the `mphantom` element is primarily intended as an aid in aligning expressions, operators adjacent to an `mphantom` should behave as if they were adjacent to the *contents* of the `mphantom`, rather than to an equivalently sized area of whitespace.

3.2.7.5 Legal grouping of space-like elements

Authors who insert space-like elements or `mphantom` elements into an existing MathML expression should note that such elements *are* counted as arguments, in elements that require a specific number of arguments, or that interpret different argument positions differently.

Therefore, space-like elements inserted into such a MathML element should be grouped with a neighboring argument of that element by introducing an `mrow` for that purpose. For example, to allow for vertical alignment on the right edge of the base of a superscript, the expression

```
<msup>
  <mi> x </mi>
  <malignmark edge="right"/>
  <mn> 2 </mn>
</msup>
```

is illegal, because `msup` must have exactly 2 arguments; the correct expression would be:

```
<msup>
  <mrow>
    <mi> x </mi>
    <malignmark edge="right"/>
  </mrow>
  <mn> 2 </mn>
</msup>
```

See also the warning about “tweaking” in [MathML-Notes].

3.2.8 String Literal `<ms>` core

3.2.8.1 Description

The `ms` element is used to represent “string literals” in expressions meant to be interpreted by computer algebra systems or other systems containing “programming languages”. By default, string literals are displayed surrounded by double quotes, with no extra spacing added around the string. As explained in 3.2.6 Text `<mtext>`, ordinary text embedded in a mathematical expression should be marked up with `mtext`, or in some cases `mo` or `mi`, but never with `ms`.

Note that the string literals encoded by `ms` are made up of characters, `mglyph`s and `malignmark`s rather than "ASCII strings". For example, `<ms>&amp;</ms>` represents a string literal containing a single character, &, and `<ms>&amp;amp;</ms>` represents a string literal containing 5 characters, the first one of which is &.

The content of `ms` elements should be rendered with visible "escaping" of certain characters in the content, including at least the left and right quoting characters, and preferably whitespace other than individual space characters. The intent is for the viewer to see that the expression is a string literal, and to see exactly which characters form its content. For example, `<ms>double quote is "</ms>` might be rendered as "double quote is \"".

Like all token elements, `ms` *does* trim and collapse whitespace in its content according to the rules of 2.1.7 Collapsing Whitespace in Input, so whitespace intended to remain in the content should be encoded as described in that section.

3.2.8.2 Attributes

`ms` elements accept the attributes listed in 3.2.2 Mathematics style attributes common to token elements, and additionally:

Name	values	default
lquote[not-core]	*string*	U+0022 (entity `quot`)
	Specifies the opening quote to enclose the content (not necessarily 'left quote' in RTL context).	
rquote[not-core]	*string*	U+0022 (entity `quot`)
	Specifies the closing quote to enclose the content (not necessarily 'right quote' in RTL context).	

3.3 General Layout Schemata

Besides tokens there are several families of MathML presentation elements. One family of elements deals with various "scripting" notations, such as subscript and superscript. Another family is concerned with matrices and tables. The remainder of the elements, discussed in this section, describe other basic notations such as fractions and radicals, or deal with general functions such as setting style properties and error handling.

3.3.1 Horizontally Group Sub-Expressions `<mrow>`[core]

3.3.1.1 Description

An `mrow` element is used to group together any number of sub-expressions, usually consisting of one or more `mo` elements acting as "operators" on one or more other expressions that are their "operands".

Several elements automatically treat their arguments as if they were contained in an `mrow` element. See the discussion of inferred `mrow`s in 3.1.3 Required Arguments. See also `mfenced` (3.3.8 Expression Inside Pair of Fences `<mfenced>`[not-core]), which can effectively form an `mrow` containing its arguments separated by commas.

`mrow` elements are typically rendered visually as a horizontal row of their arguments, left to right in the order in which the arguments occur within a context with LTR directionality, or right to left within a context with RTL directionality. The `dir` attribute can be used to specify the directionality for a specific `mrow`, otherwise it inherits the directionality from the context. For aural agents, the arguments would be rendered audibly as a sequence of renderings of the arguments. The description in 3.2.5 Operator, Fence, Separator or Accent `<mo>` of suggested rendering rules for `mo` elements assumes that all horizontal spacing between operators and their operands is added by the rendering of `mo` elements (or, more generally, embellished operators), not by the rendering of the `mrow`s they are contained in.

MathML provides support for both automatic and manual linebreaking of expressions (that is, to break excessively long expressions into several lines). All such linebreaks take place within `mrow`s, whether they are explicitly marked up in the document, or inferred (see 3.1.3.1 Inferred `<mrow>`s), although the control of linebreaking is effected through attributes on other elements (see 3.1.7 Linebreaking of Expressions).

3.3.1.2 Attributes

`mrow` elements accept the attribute listed below in addition to those listed in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
dir[core]	"ltr" | "rtl"	*inherited*
	specifies the overall directionality `ltr` (Left To Right) or `rtl` (Right To Left) to use to layout the children of the row. See 3.1.5.1 Overall Directionality of Mathematics Formulas for further discussion.	

3.3.1.3 Proper grouping of sub-expressions using `<mrow>`

Sub-expressions should be grouped by the document author in the same way as they are grouped in the mathematical interpretation of the expression; that is, according to the underlying “syntax tree” of the expression. Specifically, operators and their mathematical arguments should occur in a single `mrow`; more than one operator should occur directly in one `mrow` only when they can be considered (in a syntactic sense) to act together on the interleaved arguments, e.g. for a single parenthesized term and its parentheses, for chains of relational operators, or for sequences of terms separated by + and −. A precise rule is given below.

Proper grouping has several purposes: it improves display by possibly affecting spacing; it allows for more intelligent linebreaking and indentation; and it simplifies possible semantic interpretation of presentation elements by computer algebra systems, and audio renderers.

Although improper grouping will sometimes result in suboptimal renderings, and will often make interpretation other than pure visual rendering difficult or impossible, any grouping of expressions using `mrow` is allowed in MathML syntax; that is,

renderers should not assume the rules for proper grouping will be followed.

3.3.1.3.1 `<mrow>` OF ONE ARGUMENT

MathML renderers are required to treat an `mrow` element containing exactly one argument as equivalent in all ways to the single argument occurring alone, provided there are no attributes on the `mrow` element. If there are attributes on the `mrow` element, no requirement of equivalence is imposed. This equivalence condition is intended to simplify the implementation of MathML-generating software such as template-based authoring tools. It directly affects the definitions of embellished operator and space-like element and the rules for determining the default value of the `form` attribute of an `mo` element; see 3.2.5 Operator, Fence, Separator or Accent `<mo>` and 3.2.7 Space `<mspace/>`. See also the discussion of equivalence of MathML expressions in D.1 MathML Conformance.

3.3.1.3.2 PRECISE RULE FOR PROPER GROUPING

A precise rule for when and how to nest sub-expressions using `mrow` is especially desirable when generating MathML automatically by conversion from other formats for displayed mathematics, such as TeX, which don't always specify how sub-expressions nest. When a precise rule for grouping is desired, the following rule should be used:

Two adjacent operators, possibly embellished, possibly separated by operands (i.e. anything other than operators), should occur in the same `mrow` only when the leading operator has an infix or prefix form (perhaps inferred), the following operator has an infix or postfix form, and the operators have the same priority in the operator dictionary (B. Operator Dictionary). In all other cases, nested `mrow`s should be used.

When forming a nested `mrow` (during generation of MathML) that includes just one of two successive operators with the forms mentioned above (which means that either operator could in principle act on the intervening operand or operands), it is necessary to decide which operator acts on those operands directly (or would do so, if they were present). Ideally, this should be determined from the original expression; for example, in conversion from an operator-precedence-based format, it would be the operator with the higher precedence.

Note that the above rule has no effect on whether any MathML expression is valid, only on the recommended way of generating MathML from other formats for displayed mathematics or directly from written notation.

(Some of the terminology used in stating the above rule is defined in 3.2.5 Operator, Fence, Separator or Accent `<mo>`.)

3.3.1.4 Examples

▸ Show Section

As an example, $2x+y-z$ should be written as:

```
<mrow>
  <mrow>
    <mn> 2 </mn>
    <mo> &#x2062;<!--InvisibleTimes--> </mo>
    <mi> x </mi>
  </mrow>
  <mo> + </mo>
  <mi> y </mi>
  <mo> − </mo>
  <mi> z </mi>
</mrow>
```

$$2x + y - z$$

The proper encoding of (x, y) furnishes a less obvious example of nesting `mrow`s:

```
<mrow>
  <mo> ( </mo>
  <mrow>
    <mi> x </mi>
    <mo> , </mo>
    <mi> y </mi>
  </mrow>
  <mo> ) </mo>
</mrow>
```

$$(x, y)$$

In this case, a nested `mrow` is required inside the parentheses, since parentheses and commas, thought of as fence and separator "operators", do not act together on their arguments.

3.3.2 Fractions `<mfrac>`[core]

3.3.2.1 Description

The `mfrac` element is used for fractions. It can also be used to mark up fraction-like objects such as binomial coefficients and Legendre symbols. The syntax for `mfrac` is

```
<mfrac> numerator denominator </mfrac>
```

The `mfrac` element sets `displaystyle` to `false`, or if it was already false increments `scriptlevel` by 1, within *numerator* and *denominator*. (See 3.1.6 Displaystyle and Scriptlevel.)

3.3.2.2 Attributes

`mfrac` elements accept the attributes listed below in addition to those listed in 3.1.9 Mathematics attributes common to presentation elements. The fraction line, if any, should be drawn using the color specified by `mathcolor`.

Name	values	default
linethicknesscore	*length* | "thin" | "medium" | "thick"	medium
	Specifies the thickness of the horizontal “fraction bar”, or “rule”. The default value is `medium`; `thin` is thinner, but visible; `thick` is thicker. The exact thickness of these is left up to the rendering agent. However, if OpenType Math fonts are available then the renderer should set `medium` to the value `MATH.MathConstants.fractionRuleThickness` (the default in MathML-Core). Note: MathML Core does only allow `<length-percentage>` values.	
numalignnot-core	"left" | "center" | "right"	center
	Specifies the alignment of the numerator over the fraction.	
denomalignnot-core	"left" | "center" | "right"	center
	Specifies the alignment of the denominator under the fraction.	
bevellednot-core	"true" | "false"	false
	Specifies whether the fraction should be displayed in a bevelled style (the numerator slightly raised, the denominator slightly lowered and both separated by a slash), rather than "build up" vertically. See below for an example.	

Thicker lines (e.g. `linethickness`="thick") might be used with nested fractions; a value of "0" renders without the bar such as for binomial coefficients.

In a RTL directionality context, the numerator leads (on the right), the denominator follows (on the left) and the diagonal line slants upwards going from right to left (see 3.1.5.1 Overall Directionality of Mathematics Formulas for clarification). Although this format is an established convention, it is not universally followed; for situations where a forward slash is desired in a RTL context, alternative markup, such as an `mo` within an `mrow` should be used.

3.3.2.3 Examples

▸ Show Section

Here is an example which makes use of different values of `linethickness`:

```
<mfrac linethickness="3px">
  <mrow>
    <mo> ( </mo>
      <mfrac linethickness="0">
        <mi> a </mi>
        <mi> b </mi>
      </mfrac>
    <mo> ) </mo>
    <mfrac>
      <mi> a </mi>
      <mi> b </mi>
    </mfrac>
  </mrow>
  <mfrac>
    <mi> c </mi>
    <mi> d </mi>
  </mfrac>
</mfrac>
```

$$\frac{\binom{a}{b}\frac{a}{b}}{\frac{c}{d}}$$

This example illustrates bevelled fractions:

```
<mfrac>
  <mn> 1 </mn>
  <mrow>
    <msup>
      <mi> x </mi>
      <mn> 3 </mn>
    </msup>
    <mo> + </mo>
    <mfrac>
      <mi> x </mi>
      <mn> 3 </mn>
    </mfrac>
  </mrow>
</mfrac>
<mo> = </mo>
<mfrac bevelled="true">
  <mn> 1 </mn>
  <mrow>
    <msup>
      <mi> x </mi>
      <mn> 3 </mn>
    </msup>
    <mo> + </mo>
    <mfrac>
      <mi> x </mi>
      <mn> 3 </mn>
    </mfrac>
  </mrow>
</mfrac>
```

$$\frac{1}{x^3+\frac{x}{3}} = 1 \Big/ x^3+\frac{x}{3}$$

A more generic example is:

```
<mfrac>
  <mrow>
    <mn> 1 </mn>
    <mo> + </mo>
    <msqrt>
      <mn> 5 </mn>
    </msqrt>
  </mrow>
  <mn> 2 </mn>
</mfrac>
```

$$\frac{1+\sqrt{5}}{2}$$

3.3.3 Radicals <msqrt>core, <mroot>core

3.3.3.1 Description

These elements construct radicals. The `msqrt` element is used for square roots, while the `mroot` element is used to draw radicals with indices, e.g. a cube root. The syntax for these elements is:

```
<msqrt> base </msqrt>
<mroot> base index </mroot>
```

The `mroot` element requires exactly 2 arguments. However, `msqrt` accepts a single argument, possibly being an inferred `mrow` of multiple children; see 3.1.3 Required Arguments. The `mroot` element increments `scriptlevel` by 2, and sets `displaystyle` to `false`, within *index*, but leaves both attributes unchanged within *base*. The `msqrt` element leaves both attributes unchanged within its argument. (See 3.1.6 Displaystyle and Scriptlevel.)

Note that in a RTL directionality, the surd begins on the right, rather than the left, along with the index in the case of `mroot`.

3.3.3.2 Attributes

`msqrt` and `mroot` elements accept the attributes listed in 3.1.9 Mathematics attributes common to presentation elements. The surd and overbar should be drawn using the color specified by `mathcolor`.

3.3.3.3 Examples

▸ Show Section

Square roots and cube roots

```
<mrow>
  <mrow>
    <msqrt>
      <mi>x</mi>
    </msqrt>
    <mroot>
      <mi>x</mi>
      <mn>3</mn>
    </mroot>
  <mrow>
  <mo>=</mo>
  <msup>
    <mi>x</mi>
    <mrow>
      <mrow>
        <mn>1</mn>
        <mo>/</mo>
        <mn>2</mn>
      </mrow>
      <mo>+</mo>
      <mrow>
        <mn>1</mn>
        <mo>/</mo>
        <mn>3</mn>
      </mrow>
    </mrow>
  </msup>
</mrow>
```

$$\sqrt{x}\sqrt[3]{x} = x^{1/2+1/3}$$

3.3.4 Style Change <mstyle>core

3.3.4.1 Description

The `mstyle` element is used to make style changes that affect the rendering of its contents. As a presentation element, it accepts the attributes described in 3.1.9 Mathematics attributes common to presentation elements. Additionally, it can be given any attribute accepted by any other presentation element, except for the attributes described below. Finally, the `mstyle` element can be given certain special attributes listed in the next subsection.

The `mstyle` element accepts a single argument, possibly being an inferred `mrow` of multiple children; see 3.1.3 Required Arguments.

Loosely speaking, the effect of the `mstyle` element is to change the default value of an attribute for the elements it contains. Style changes work in one of several ways, depending on the way in which default values are specified for an attribute. The

cases are:

- Some attributes, such as `displaystyle` or `scriptlevel` (explained below), are inherited from the surrounding context when they are not explicitly set. Specifying such an attribute on an `mstyle` element sets the value that will be inherited by its child elements. Unless a child element overrides this inherited value, it will pass it on to its children, and they will pass it to their children, and so on. But if a child element does override it, either by an explicit attribute setting or automatically (as is common for `scriptlevel`), the new (overriding) value will be passed on to that element's children, and then to their children, etc, unless it is again overridden.
- Other attributes, such as `linethickness` on `mfrac`, have default values that are not normally inherited. That is, if the `linethickness` attribute is not set on the `mfrac` element, it will normally use the default value of `medium`, even if it was contained in a larger `mfrac` element that set this attribute to a different value. For attributes like this, specifying a value with an `mstyle` element has the effect of changing the default value for all elements within its scope. The net effect is that setting the attribute value with `mstyle` propagates the change to all the elements it contains directly or indirectly, except for the individual elements on which the value is overridden. Unlike in the case of inherited attributes, elements that explicitly override this attribute have no effect on this attribute's value in their children.
- Another group of attributes, such as `stretchy` and `form`, are computed from operator dictionary information, position in the enclosing `mrow`, and other similar data. For these attributes, a value specified by an enclosing `mstyle` overrides the value that would normally be computed.

Note that attribute values inherited from an `mstyle` in any manner affect a descendant element in the `mstyle`'s content only if that attribute is not given a value by the descendant element. On any element for which the attribute is set explicitly, the value specified overrides the inherited value. The only exception to this rule is when the attribute value is documented as specifying an incremental change to the value inherited from that element's context or rendering environment.

Note also that the difference between inherited and non-inherited attributes set by `mstyle`, explained above, only matters when the attribute is set on some element within the `mstyle`'s contents that has descendants also setting it. Thus it never matters for attributes, such as `mathsize`, which can only be set on token elements (or on `mstyle` itself).

MathML specifies that when the attributes `height`, `depth` or `width` are specified on an `mstyle` element, they apply only to `mspace` elements, and not to the corresponding attributes of `mglyph`, `mpadded`, or `mtable`. Similarly, when `rowalign` or `columnalign` are specified on an `mstyle` element, they apply only to the `mtable` element, and not the `mtr`, `mtd`, and `maligngroup` elements. When the `lspace` attribute is set with `mstyle`, it applies only to the `mo` element and not to `mpadded`. To be consistent, the `voffset` attribute of the `mpadded` element can not be set on `mstyle`. When the `align` attribute is set with `mstyle`, it applies only to the `munder`, `mover`, and `munderover` elements, and not to the `mtable` and `mstack` elements. The attributes `src` and `alt` on `mglyph`, and `actiontype` on `maction`, cannot be set on `mstyle`.

As a presentation element, `mstyle` directly accepts the `mathcolor` and `mathbackground` attributes. Thus, the `mathbackground` specifies the color to fill the bounding box of the `mstyle` element itself; it does *not* specify the default background color. This is an incompatible change from MathML 2, but it is more useful and intuitive. Since the default for `mathcolor` is inherited, this is no change in its behavior.

§ *3.3.4.2 Attributes*

As stated above, `mstyle` accepts all attributes of all MathML presentation elements which do not have required values. That is, all attributes which have an explicit default value or a default value which is inherited or computed are accepted by the `mstyle` element. This group of attributes is *not* accepted in MathML Core.

`mstyle` elements accept the attributes listed in 3.1.9 Mathematics attributes common to presentation elements.

Additionally, `mstyle` can be given the following special attributes that are implicitly inherited by every MathML element as part of its rendering environment:

Name	values	default
scriptlevel[core]	("+" | "-")? *unsigned-integer*	*inherited*
	Changes the `scriptlevel` in effect for the children. When the value is given without a sign, it sets `scriptlevel` to the specified value; when a sign is given, it increments ("+") or decrements ("-") the current value. (Note that large decrements can result in negative values of `scriptlevel`, but these values are considered legal.) See 3.1.6 Displaystyle and Scriptlevel.	
displaystyle[core]	"true" | "false"	*inherited*
	Changes the `displaystyle` in effect for the children. See 3.1.6 Displaystyle and Scriptlevel.	
scriptsizemultiplier[not-core]	*number*	0.71
	Specifies the multiplier to be used to adjust font size due to changes in `scriptlevel`. See 3.1.6 Displaystyle and Scriptlevel.	
scriptminsize[not-core]	*length*	8pt
	Specifies the minimum font size allowed due to changes in `scriptlevel`. Note that this does not limit the font size due to changes to `mathsize`. See 3.1.6 Displaystyle and Scriptlevel.	
infixlinebreakstyle[not-core]	"before" | "after" | "duplicate"	before
	Specifies the default linebreakstyle to use for infix operators; see 3.2.5.2.2 Linebreaking attributes	
decimalpoint[not-core]	*character*	.
	Specifies the character used to determine the alignment point within `mstack` and `mtable` columns when the "decimalpoint" value is used to specify the alignment. The default, ".", is the decimal separator used to separate the integral and decimal fractional parts of floating point numbers in many countries. (See 3.6 Elementary Math and 3.5.4 Alignment Markers `<maligngroup/>`, `<malignmark/>`[not-core]).	

If `scriptlevel` is changed incrementally by an `mstyle` element that also sets certain other attributes, the overall effect of the changes may depend on the order in which they are processed. In such cases, the attributes in the following list should be processed in the following order, regardless of the order in which they occur in the XML-format attribute list of the `mstyle` start tag: `scriptsizemultiplier`, `scriptminsize`, `scriptlevel`, `mathsize`.

3.3.4.3 Examples

▶ Show Section

In a continued fraction, the nested fractions should not shrink. Instead, they should remain the same size. This can be accomplished by resetting `displaystyle` and `scriptlevel` for the children of each `mfrac` using `mstyle` as shown below:

```
<mrow>
  <mi>π</mi>
  <mo>=</mo>
  <mfrac>
    <mstyle displaystyle="true" scriptlevel="0"> <mn>4</mn> </mstyle>
    <mstyle displaystyle="true" scriptlevel="0">
      <mn>1</mn>
      <mo>+</mo>
      <mfrac>
        <mstyle displaystyle="true" scriptlevel="0">
          <msup> <mn>1</mn> <mn>2</mn> </msup>
        </mstyle>
        <mstyle displaystyle="true" scriptlevel="0">
          <mn>2</mn>
          <mo>+</mo>
          <mfrac>
            <mstyle displaystyle="true" scriptlevel="0">
              <msup> <mn>3</mn> <mn>2</mn> </msup>
            </mstyle>
            <mstyle displaystyle="true" scriptlevel="0">
              <mn>2</mn>
              <mo>+</mo>
              <mfrac>
                <mstyle displaystyle="true" scriptlevel="0">
                  <msup> <mn>5</mn> <mn>2</mn> </msup>
                </mstyle>
                <mstyle displaystyle="true" scriptlevel="0">
                  <mn>2</mn>
                  <mo>+</mo>
                  <mfrac>
                    <mstyle displaystyle="true" scriptlevel="0">
                      <msup> <mn>7</mn> <mn>2</mn> </msup>
                    </mstyle>
                    <mstyle displaystyle="true" scriptlevel="0">
                      <mn>2</mn>
                      <mo>+</mo>
                      <mo>⋱</mo>
                    </mstyle>
                  </mfrac>
                </mstyle>
              </mfrac>
            </mstyle>
          </mfrac>
        </mstyle>
      </mfrac>
    </mstyle>
  </mfrac>
</mrow>
```

$$\pi = \cfrac{4}{1+\cfrac{1^2}{2+\cfrac{3^2}{2+\cfrac{5^2}{2+\cfrac{7^2}{2+\ddots}}}}}$$

3.3.5 Error Message <merror>[core]

3.3.5.1 Description

The `merror` element displays its contents as an "error message". This might be done, for example, by displaying the contents in red, flashing the contents, or changing the background color. The contents can be any expression or expression sequence.

`merror` accepts a single argument possibly being an inferred `mrow` of multiple children; see 3.1.3 Required Arguments.

The intent of this element is to provide a standard way for programs that *generate* MathML from other input to report syntax errors in their input. Since it is anticipated that preprocessors that parse input syntaxes designed for easy hand entry will be developed to generate MathML, it is important that they have the ability to indicate that a syntax error occurred at a certain point. See D.2 Handling of Errors.

The suggested use of `merror` for reporting syntax errors is for a preprocessor to replace the erroneous part of its input with an `merror` element containing a description of the error, while processing the surrounding expressions normally as far as possible. By this means, the error message will be rendered where the erroneous input would have appeared, had it been correct; this makes it easier for an author to determine from the rendered output what portion of the input was in error.

No specific error message format is suggested here, but as with error messages from any program, the format should be designed to make as clear as possible (to a human viewer of the rendered error message) what was wrong with the input and how it can be fixed. If the erroneous input contains correctly formatted subsections, it may be useful for these to be preprocessed normally and included in the error message (within the contents of the `merror` element), taking advantage of the ability of `merror` to contain arbitrary MathML expressions rather than only text.

3.3.5.2 Attributes

`merror` elements accept the attributes listed in 3.1.9 Mathematics attributes common to presentation elements.

3.3.5.3 Example

▶ Show Section

If a MathML syntax-checking preprocessor received the input

```
<mfraction>
  <mrow> <mn> 1 </mn> <mo> + </mo> <msqrt> <mn> 5 </mn> </msqrt> </mrow>
  <mn> 2 </mn>
</mfraction>
```

which contains the non-MathML element `mfraction` (presumably in place of the MathML element `mfrac`), it might generate the error message

```
<merror>
  <mtext> Unrecognized element: mfraction; arguments were: </mtext>
  <mrow> <mn> 1 </mn> <mo> + </mo> <msqrt> <mn> 5 </mn> </msqrt> </mrow>
  <mtext> and </mtext>
  <mn> 2 </mn>
</merror>
```

Unrecognized element: mfraction; arguments were: $1 + \sqrt{5}$ and 2

Note that the preprocessor's input is not, in this case, valid MathML, but the error message it outputs is valid MathML.

3.3.6 Adjust Space Around Content <mpadded>[core]

3.3.6.1 Description

An `mpadded` element renders the same as its child content, but with the size of the child's bounding box and the relative positioning point of its content modified according to `mpadded`'s attributes. It does not rescale (stretch or shrink) its content. The name of the element reflects the typical use of `mpadded` to add padding, or extra space, around its content. However, `mpadded` can be used to make more general adjustments of size and positioning, and some combinations, e.g. negative padding, can cause the content of `mpadded` to overlap the rendering of neighboring content. See [MathML-Notes] for warnings about several potential pitfalls of this effect.

The `mpadded` element accepts a single argument which may be an inferred `mrow` of multiple children; see 3.1.3 Required Arguments.

It is suggested that audio renderers add (or shorten) time delays based on the attributes representing horizontal space (`width` and `lspace`).

3.3.6.2 Attributes

`mpadded` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common

to presentation elements.

Name	values	default
heightcore	*length*	*same as content*
	Sets or increments the height of the `mpadded` element. See below for discussion.	
depthcore	*length*	*same as content*
	Sets or increments the depth of the `mpadded` element. See below for discussion.	
widthcore	*length*	*same as content*
	Sets or increments the width of the `mpadded` element. See below for discussion.	
lspacecore	*length*	0em
	Sets the horizontal position of the child content. See below for discussion.	
voffsetcore	*length*	0em
	Sets the vertical position of the child content. See below for discussion.	

NOTE: mpadded lengths in MathML 3

While [MathML-Core] supports the above attributes, it only allows the value to be a valid `<length-percentage>`. As described in *length* MathML 4 extends this syntax to allow namedspace.

MathML 3 also allowed additional extensions:

- A leading "+" or "-" denoted a relative increment or decrement from the default value. This is not supported however the same fuunctionality is now available in standard CSS `<length-percentage>` syntax: `height="calc(100%+10pt)"`.
- MathML 3 also specified the *pseudo-units* `height`, `depth` and `width`. These are not supported in MathML 4 however the main use cases are addressed using percentage values, `height="0.5height"` is equivalent to `height="50%`.

These attributes specify the size of the bounding box of the `mpadded` element relative to the size of the bounding box of its child content, and specify the position of the child content of the `mpadded` element relative to the natural positioning of the `mpadded` element. The typographical layout parameters determined by these attributes are described in the next subsection. Depending on the form of the attribute value, a dimension may be set to a new value, or specified relative to the child content's corresponding dimension. Values may be given as multiples or percentages of any of the dimensions of the normal rendering of the child content using so-called pseudo-units, or they can be set directly using standard units, see 2.1.5.2 Length Valued Attributes.

The corresponding dimension is set to the following length value. specifying a length that would produce a net negative value for these attributes has the same effect as setting the attribute to zero. In other words, the effective bounding box of an `mpadded` element always has non-negative dimensions. However, negative values are allowed for the relative positioning

attributes `lspace` and `voffset`.

3.3.6.3 Meanings of size and position attributes

The content of an `mpadded` element defines a fragment of mathematical notation, such as a character, fraction, or expression, that can be regarded as a single typographical element with a natural positioning point relative to its natural bounding box.

The size of the bounding box of an `mpadded` element is defined as the size of the bounding box of its content, except as modified by the `mpadded` element's `height`, `depth`, and `width` attributes. The natural positioning point of the child content of the `mpadded` element is located to coincide with the natural positioning point of the `mpadded` element, except as modified by the `lspace` and `voffset` attributes. Thus, the size attributes of `mpadded` can be used to expand or shrink the apparent bounding box of its content, and the position attributes of `mpadded` can be used to move the content relative to the bounding box (and hence also neighboring elements). Note that MathML doesn't define the precise relationship between "ink", bounding boxes and positioning points, which are implementation specific. Thus, absolute values for mpadded attributes may not be portable between implementations.

The `height` attribute specifies the vertical extent of the bounding box of the `mpadded` element above its baseline. Increasing the `height` increases the space between the baseline of the `mpadded` element and the content above it, and introduces padding above the rendering of the child content. Decreasing the `height` reduces the space between the baseline of the `mpadded` element and the content above it, and removes space above the rendering of the child content. Decreasing the `height` may cause content above the `mpadded` element to overlap the rendering of the child content, and should generally be avoided.

The `depth` attribute specifies the vertical extent of the bounding box of the `mpadded` element below its baseline. Increasing the `depth` increases the space between the baseline of the `mpadded` element and the content below it, and introduces padding below the rendering of the child content. Decreasing the `depth` reduces the space between the baseline of the `mpadded` element and the content below it, and removes space below the rendering of the child content. Decreasing the `depth` may cause content below the `mpadded` element to overlap the rendering of the child content, and should generally be avoided.

The `width` attribute specifies the horizontal distance between the positioning point of the `mpadded` element and the positioning point of the following content. Increasing the `width` increases the space between the positioning point of the `mpadded` element and the content that follows it, and introduces padding after the rendering of the child content. Decreasing the `width` reduces the space between the positioning point of the `mpadded` element and the content that follows it, and removes space after the rendering of the child content. Setting the `width` to zero causes following content to be positioned at the positioning point of the `mpadded` element. Decreasing the `width` should generally be avoided, as it may cause overprinting of the following content.

The `lspace` attribute ("leading" space; see 3.1.5.1 Overall Directionality of Mathematics Formulas) specifies the horizontal location of the positioning point of the child content with respect to the positioning point of the `mpadded` element. By default they coincide, and therefore absolute values for lspace have the same effect as relative values. Positive values for the `lspace` attribute increase the space between the preceding content and the child content, and introduce padding before the rendering of the child content. Negative values for the `lspace` attributes reduce the space between the preceding content and the child content, and may cause overprinting of the preceding content, and should generally be avoided. Note that the `lspace` attribute does not affect the `width` of the `mpadded` element, and so the `lspace` attribute will also affect the space between the child content and following content, and may cause overprinting of the following content, unless the `width` is

adjusted accordingly.

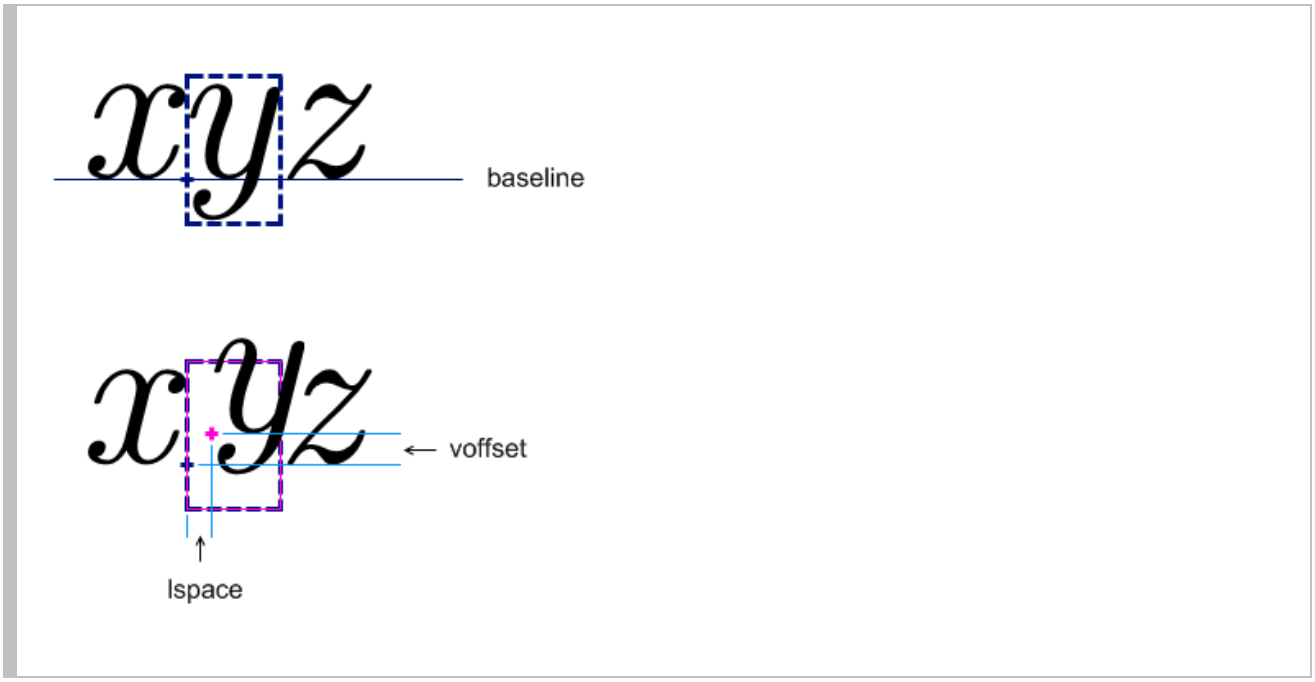
The `voffset` attribute specifies the vertical location of the positioning point of the child content with respect to the positioning point of the `mpadded` element. Positive values for the `voffset` attribute raise the rendering of the child content above the baseline. Negative values for the `voffset` attribute lower the rendering of the child content below the baseline. In either case, the `voffset` attribute may cause overprinting of neighboring content, which should generally be avoided. Note that the `voffset` attribute does not affect the `height` or `depth` of the `mpadded` element, and so the `voffset` attribute will also affect the space between the child content and neighboring content, and may cause overprinting of the neighboring content, unless the `height` or `depth` is adjusted accordingly.

MathML renderers should ensure that, except for the effects of the attributes, the relative spacing between the contents of the `mpadded` element and surrounding MathML elements would not be modified by replacing an `mpadded` element with an `mrow` element with the same content, even if linebreaking occurs within the `mpadded` element. MathML does not define how non-default attribute values of an `mpadded` element interact with the linebreaking algorithm.

§ *3.3.6.4 Examples*

▸ Show Section

The effects of the size and position attributes are illustrated below. The following diagram illustrates the use of `lspace` and `voffset` to shift the position of child content without modifying the `mpadded` bounding box.

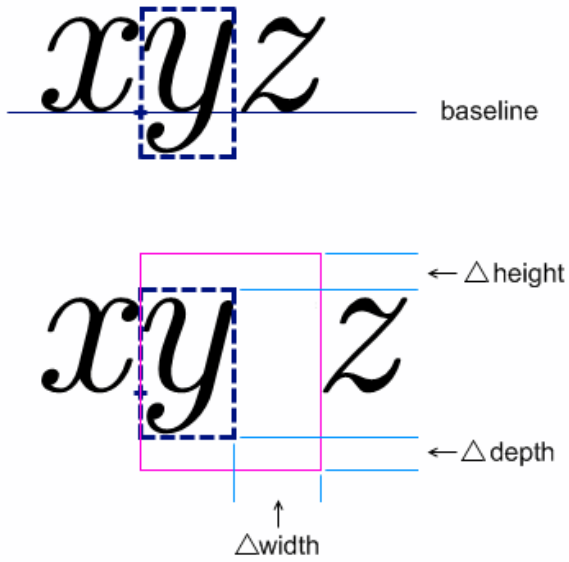


The corresponding MathML is:

```
<mrow>
  <mi>x</mi>
  <mpadded lspace="0.2em" voffset="0.3ex">
    <mi>y</mi>
  </mpadded>
  <mi>z</mi>
</mrow>
```

$$x\,yz$$

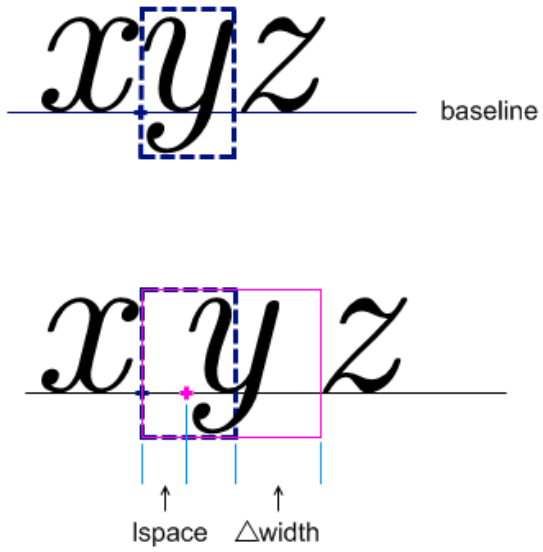
The next diagram illustrates the use of `width`, `height` and `depth` to modifying the `mpadded` bounding box without changing the relative position of the child content.



The corresponding MathML is:

```
<mrow>
  <mi>x</mi>
  <mpadded width="190%" height="calc(100% +0.3ex)" depth="calc(100% +0.3ex)">
    <mi>y</mi>
  </mpadded>
  <mi>z</mi>
</mrow>
```

The final diagram illustrates the generic use of `mpadded` to modify both the bounding box and relative position of child content.



The corresponding MathML is:

```
<mrow>
  <mi>x</mi>
  <mpadded lspace="0.3em" width="calc(100% +0.6em)">
    <mi>y</mi>
  </mpadded>
  <mi>z</mi>
</mrow>
```

3.3.7 Making Sub-Expressions Invisible <mphantom>[core]

3.3.7.1 Description

The `mphantom` element renders invisibly, but with the same size and other dimensions, including baseline position, that its contents would have if they were rendered normally. `mphantom` can be used to align parts of an expression by invisibly duplicating sub-expressions.

The `mphantom` element accepts a single argument possibly being an inferred `mrow` of multiple children; see 3.1.3 Required Arguments.

Note that it is possible to wrap both an `mphantom` and an `mpadded` element around one MathML expression, as in `<mphantom><mpadded attribute-settings> ... </mpadded></mphantom>`, to change its size and make it invisible at the same time.

MathML renderers should ensure that the relative spacing between the contents of an `mphantom` element and the

surrounding MathML elements is the same as it would be if the `mphantom` element were replaced by an `mrow` element with the same content. This holds even if linebreaking occurs within the `mphantom` element.

For the above reason, `mphantom` is *not* considered space-like (3.2.7 Space <mspace/>) unless its content is space-like, since the suggested rendering rules for operators are affected by whether nearby elements are space-like. Even so, the warning about the legal grouping of space-like elements may apply to uses of `mphantom`.

3.3.7.2 Attributes

`mphantom` elements accept the attributes listed in 3.1.9 Mathematics attributes common to presentation elements (the `mathcolor` has no effect).

3.3.7.3 Examples

▸ Show Section

There is one situation where the preceding rules for rendering an `mphantom` may not give the desired effect. When an `mphantom` is wrapped around a subsequence of the arguments of an `mrow`, the default determination of the `form` attribute for an `mo` element within the subsequence can change. (See the default value of the `form` attribute described in 3.2.5 Operator, Fence, Separator or Accent <mo>.) It may be necessary to add an explicit `form` attribute to such an `mo` in these cases. This is illustrated in the following example.

In this example, `mphantom` is used to ensure alignment of corresponding parts of the numerator and denominator of a fraction:

```
<mfrac>
  <mrow>
    <mi> x </mi>
    <mo> + </mo>
    <mi> y </mi>
    <mo> + </mo>
    <mi> z </mi>
  </mrow>
  <mrow>
    <mi> x </mi>
    <mphantom>
      <mo form="infix"> + </mo>
      <mi> y </mi>
    </mphantom>
    <mo> + </mo>
    <mi> z </mi>
  </mrow>
</mfrac>
```

$$\frac{x+y+z}{x\quad\ +z}$$

This would render as something like

$$\frac{x+y+x}{x\quad\ +z}$$

rather than as

$$\frac{x+y+z}{x+z}$$

The explicit attribute setting `form="infix"` on the `mo` element inside the `mphantom` sets the `form` attribute to what it would have been in the absence of the surrounding `mphantom`. This is necessary since otherwise, the `+` sign would be interpreted as a prefix operator, which might have slightly different spacing.

Alternatively, this problem could be avoided without any explicit attribute settings, by wrapping each of the arguments `<mo>+</mo>` and `<mi>y</mi>` in its own `mphantom` element, i.e.

```
<mfrac>
  <mrow>
    <mi> x </mi>
    <mo> + </mo>
    <mi> y </mi>
    <mo> + </mo>
    <mi> z </mi>
  </mrow>
  <mrow>
    <mi> x </mi>
    <mphantom>
      <mo> + </mo>
    </mphantom>
    <mphantom>
      <mi> y </mi>
    </mphantom>
    <mo> + </mo>
    <mi> z </mi>
  </mrow>
</mfrac>
```

$$\frac{x+y+z}{x\quad\ +z}$$

3.3.8 Expression Inside Pair of Fences `<mfenced>` not-core

3.3.8.1 Description

The `mfenced` element provides a convenient form in which to express common constructs involving fences (i.e. braces, brackets, and parentheses), possibly including separators (such as comma) between the arguments.

For example, `<mfenced> <mi>x</mi> </mfenced>` renders as "(x)" and is equivalent to

```
<mrow> <mo> ( </mo> <mi>x</mi> <mo> ) </mo> </mrow>
```

$$(x)$$

and `<mfenced> <mi>x</mi> <mi>y</mi> </mfenced>` renders as "(x, y)" and is equivalent to

```
<mrow>
  <mo> ( </mo>
  <mrow> <mi>x</mi> <mo>,</mo> <mi>y</mi> </mrow>
  <mo> ) </mo>
</mrow>
```

$$(x, y)$$

Individual fences or separators are represented using `mo` elements, as described in 3.2.5 Operator, Fence, Separator or Accent `<mo>`. Thus, any `mfenced` element is completely equivalent to an expanded form described below. While `mfenced` might be more convenient for authors or authoring software, only the expanded form is supported in [MathML-Core]. A renderer that supports this recommendation is required to render either of these forms in exactly the same way.

In general, an `mfenced` element can contain zero or more arguments, and will enclose them between fences in an `mrow`; if there is more than one argument, it will insert separators between adjacent arguments, using an additional nested `mrow` around the arguments and separators for proper grouping (3.3.1 Horizontally Group Sub-Expressions `<mrow>`). The general expanded form is shown below. The fences and separators will be parentheses and comma by default, but can be changed using attributes, as shown in the following table.

3.3.8.2 Attributes

`mfenced` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements. The delimiters and separators should be drawn using the color specified by `mathcolor`.

Name	values	default
open[not-core]	*string*	(
	Specifies the opening delimiter. Since it is used as the content of an `mo` element, any whitespace will be trimmed and collapsed as described in 2.1.7 Collapsing Whitespace in Input.	
close[not-core]	*string*	)
	Specifies the closing delimiter. Since it is used as the content of an `mo` element, any whitespace will be trimmed and collapsed as described in 2.1.7 Collapsing Whitespace in Input.	
separators[not-core]	*string*	,
	Specifies a sequence of zero or more separator characters, optionally separated by whitespace. Each pair of arguments is displayed separated by the corresponding separator (none appears after the last argument). If there are too many separators, the excess are ignored; if there are too few, the last separator is repeated. Any whitespace within `separators` is ignored.	

A generic `mfenced` element, with all attributes explicit, looks as follows:

```
<mfenced open="opening-fence"
         close="closing-fence"
         separators="sep#1 sep#2 ... sep#(n-1)" >
  arg#1
  ...
  arg#n
</mfenced>
```

In an RTL directionality context, since the initial text direction is RTL, characters in the `open` and `close` attributes that have a mirroring counterpart will be rendered in that mirrored form. In particular, the default values will render correctly as a parenthesized sequence in both LTR and RTL contexts.

The general `mfenced` element shown above is equivalent to the following expanded form:

```
<mrow>
  <mo fence="true"> opening-fence </mo>
  <mrow>
    arg#1
    <mo separator="true"> sep#1 </mo>
    ...
    <mo separator="true"> sep#(n-1) </mo>
    arg#n
  </mrow>
  <mo fence="true"> closing-fence </mo>
</mrow>
```

Each argument except the last is followed by a separator. The inner `mrow` is added for proper grouping, as described in 3.3.1

Horizontally Group Sub-Expressions <mrow>.

When there is only one argument, the above form has no separators; since <mrow> arg#1 </mrow> is equivalent to arg#1 (as described in 3.3.1 Horizontally Group Sub-Expressions <mrow>), this case is also equivalent to:

```
<mrow>
  <mo fence="true"> opening-fence </mo>
    arg#1
  <mo fence="true"> closing-fence </mo>
</mrow>
```

If there are too many separator characters, the extra ones are ignored. If separator characters are given, but there are too few, the last one is repeated as necessary. Thus, the default value of separators="," is equivalent to separators=",,", separators=",,,", etc. If there are no separator characters provided but some are needed, for example if separators=" " or "" and there is more than one argument, then no separator elements are inserted at all — that is, the elements <mo separator="true"> sep#i </mo> are left out entirely. Note that this is different from inserting separators consisting of mo elements with empty content.

Finally, for the case with no arguments, i.e.

```
<mfenced open="opening-fence"
 close="closing-fence"
 separators="anything" >
</mfenced>
```

the equivalent expanded form is defined to include just the fences within an mrow:

```
<mrow>
  <mo fence="true"> opening-fence </mo>
  <mo fence="true"> closing-fence </mo>
</mrow>
```

Note that not all “fenced expressions” can be encoded by an mfenced element. Such exceptional expressions include those with an “embellished” separator or fence or one enclosed in an mstyle element, a missing or extra separator or fence, or a separator with multiple content characters. In these cases, it is necessary to encode the expression using an appropriately modified version of an expanded form. As discussed above, it is always permissible to use the expanded form directly, even when it is not necessary. In particular, authors cannot be guaranteed that MathML preprocessors won't replace occurrences of mfenced with equivalent expanded forms.

Note that the equivalent expanded forms shown above include attributes on the mo elements that identify them as fences or separators. Since the most common choices of fences and separators already occur in the operator dictionary with those attributes, authors would not normally need to specify those attributes explicitly when using the expanded form directly. Also, the rules for the default form attribute (3.2.5 Operator, Fence, Separator or Accent <mo>) cause the opening and closing fences to be effectively given the values form="prefix" and form="postfix" respectively, and the separators to be given the value form="infix".

Note that it would be incorrect to use mfenced with a separator of, for instance, “+”, as an abbreviation for an expression using “+” as an ordinary operator, e.g.

```
<mrow>
  <mi>x</mi> <mo>+</mo> <mi>y</mi> <mo>+</mo> <mi>z</mi>
</mrow>
```

$$x + y + z$$

This is because the + signs would be treated as separators, not infix operators. That is, it would render as if they were marked up as `<mo separator="true">+</mo>`, which might therefore render inappropriately.

3.3.8.3 Examples

▸ Show Section

```
<mfenced>
  <mrow>
    <mi> a </mi>
    <mo> + </mo>
    <mi> b </mi>
  </mrow>
</mfenced>
```

Note that the above `mrow` is necessary so that the `mfenced` has just one argument. Without it, this would render incorrectly as "$(a, +, b)$".

```
<mfenced open="[">
  <mn> 0 </mn>
  <mn> 1 </mn>
</mfenced>
```

```
<mrow>
  <mi> f </mi>
  <mo> &#x2061;<!--ApplyFunction--> </mo>
  <mfenced>
    <mi> x </mi>
    <mi> y </mi>
  </mfenced>
</mrow>
```

3.3.9 Enclose Expression Inside Notation `<menclose>`[not-core]

3.3.9.1 Description

The `menclose` element renders its content inside the enclosing notation specified by its `notation` attribute. `menclose` accepts a single argument possibly being an inferred `mrow` of multiple children; see 3.1.3 Required Arguments.

3.3.9.2 Attributes

`menclose` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements. The notations should be drawn using the color specified by `mathcolor`.

The values allowed for `notation` are open-ended. Conforming renderers may ignore any value they do not handle, although renderers are encouraged to render as many of the values listed below as possible.

Name	values	default
notation[not-core]	(`actuarial` | `phasorangle` | `box` | `roundedbox` | `circle` | `left` | `right` | `top` | `bottom` | `updiagonalstrike` | `downdiagonalstrike` | `verticalstrike` | `horizontalstrike` | `northeastarrow` | `madruwb` | text) +	do nothing
	Specifies a space separated list of notations to be used to enclose the children. See below for a description of each type of notation. MathML 4 deprecates the use of `longdiv` and `radical`. These notations duplicate functionality provided by `mlongdiv` and `msqrt` respectively; those elements should be used instead. The default has been changed so that if no `notation` is given, or if it is an empty string, then `menclose` should not draw.	

Any number of values can be given for `notation` separated by whitespace; all of those given and understood by a MathML renderer should be rendered. Each should be rendered as if the others were not present; they should not nest one inside of the other. For example, `notation="circle box"` should result in circle and a box around the contents of `menclose`; the circle and box may overlap. This is shown in the first example below. Of the predefined notations, only `phasorangle` is affected by the directionality (see 3.1.5.1 Overall Directionality of Mathematics Formulas):

When `notation` is specified as `actuarial`, the contents are drawn enclosed by an actuarial symbol. A similar result can be achieved with the value `top right`.

The values `box`, `roundedbox`, and `circle` should enclose the contents as indicated by the values. The amount of distance between the box, roundedbox, or circle, and the contents are not specified by MathML, and left to the renderer. In practice, paddings on each side of 0.4em in the horizontal direction and .5ex in the vertical direction seem to work well.

The values `left`, `right`, `top` and `bottom` should result in lines drawn on those sides of the contents. The values `northeastarrow`, `updiagonalstrike`, `downdiagonalstrike`, `verticalstrike` and `horizontalstrike` should result in the indicated strikeout lines being superimposed over the content of the `menclose`, e.g. a strikeout that extends from the lower left corner to the upper right corner of the `menclose` element for `updiagonalstrike`, etc.

The value `northeastarrow` is a recommended value to implement because it can be used to implement TeX's \cancelto command. If a renderer implements other arrows for `menclose`, it is recommended that the arrow names are chosen from

the following full set of names for consistency and standardization among renderers:

- `uparrow`
- `rightarrow`
- `downarrow`
- `leftarrow`
- `northwestarrow`
- `southwestarrow`
- `southeastarrow`
- `northeastarrow`
- `updownarrow`
- `leftrightarrow`
- `northwestsoutheastarrow`
- `northeastsouthwestarrow`

The value `madruwb` should generate an enclosure representing an Arabic factorial ('madruwb' is the transliteration of the Arabic مضروب for factorial). This is shown in the third example below.

The baseline of an `menclose` element is the baseline of its child (which might be an implied `mrow`).

3.3.9.3 Examples

▸ Show Section

An example of using multiple attributes is

```
<menclose notation='circle box'>
  <mi> x </mi><mo> + </mo><mi> y </mi>
</menclose>
```

An example of using `menclose` for actuarial notation is

```
<msub>
  <mi>a</mi>
  <mrow>
    <menclose notation='actuarial'>
      <mi>n</mi>
    </menclose>
    <mo>&#x2063;<!--InvisibleComma--></mo>
    <mi>i</mi>
  </mrow>
</msub>
```

$a_{\overline{n}|i}$

An example of `phasorangle`, which is used in circuit analysis, is:

```
<mi>C</mi>
<mrow>
  <menclose notation='phasorangle'>
    <mrow>
      <mo>−</mo>
      <mfrac>
        <mi>π</mi>
        <mn>2</mn>
      </mfrac>
    </mrow>
  </menclose>
</mrow>
```

$C\angle -\frac{\pi}{2}$

An example of `madruwb` is:

```
<menclose notation="madruwb">
  <mn>12</mn>
</menclose>
```

12

3.4 Script and Limit Schemata

The elements described in this section position one or more scripts around a base. Attaching various kinds of scripts and

embellishments to symbols is a very common notational device in mathematics. For purely visual layout, a single general-purpose element could suffice for positioning scripts and embellishments in any of the traditional script locations around a given base. However, in order to capture the abstract structure of common notation better, MathML provides several more specialized scripting elements.

In addition to sub-/superscript elements, MathML has overscript and underscript elements that place scripts above and below the base. These elements can be used to place limits on large operators, or for placing accents and lines above or below the base. The rules for rendering accents differ from those for overscripts and underscripts, and this difference can be controlled with the `accent` and `accentunder` attributes, as described in the appropriate sections below.

Rendering of scripts is affected by the `scriptlevel` and `displaystyle` attributes, which are part of the environment inherited by the rendering process of every MathML expression, and are described in 3.1.6 Displaystyle and Scriptlevel. These attributes cannot be given explicitly on a scripting element, but can be specified on the start tag of a surrounding `mstyle` element if desired.

MathML also provides an element for attachment of tensor indices. Tensor indices are distinct from ordinary subscripts and superscripts in that they must align in vertical columns. Also, all the upper scripts should be baseline-aligned and all the lower scripts should be baseline-aligned. Tensor indices can also occur in prescript positions. Note that ordinary scripts follow the base (on the right in LTR context, but on the left in RTL context); prescripts precede the base (on the left (right) in LTR (RTL) context).

Because presentation elements should be used to describe the abstract notational structure of expressions, it is important that the base expression in all "scripting" elements (i.e. the first argument expression) should be the entire expression that is being scripted, not just the trailing character. For example, $(x+y)^2$ should be written as:

```
<msup>
  <mrow>
    <mo> ( </mo>
    <mrow>
      <mi> x </mi>
      <mo> + </mo>
      <mi> y </mi>
    </mrow>
    <mo> ) </mo>
  </mrow>
  <mn> 2 </mn>
</msup>
```

$$(x+y)^2$$

3.4.1 Subscript `<msub>`[core]

3.4.1.1 Description

The `msub` element attaches a subscript to a base using the syntax

```
<msub> base subscript </msub>
```

It increments `scriptlevel` by 1, and sets `displaystyle` to `false`, within *subscript*, but leaves both attributes unchanged within *base*. (See 3.1.6 Displaystyle and Scriptlevel.)

3.4.1.2 Attributes

`msub` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
subscriptshift[not-core]	*length*	*automatic*
	Specifies the minimum amount to shift the baseline of *subscript* down; the default is for the rendering agent to use its own positioning rules.	

3.4.2 Superscript `<msup>`[core]

3.4.2.1 Description

The `msup` element attaches a superscript to a base using the syntax

```
<msup> base superscript </msup>
```

It increments `scriptlevel` by 1, and sets `displaystyle` to `false`, within *superscript*, but leaves both attributes unchanged within *base*. (See 3.1.6 Displaystyle and Scriptlevel.)

3.4.2.2 Attributes

`msup` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
superscriptshift[not-core]	*length*	*automatic*
	Specifies the minimum amount to shift the baseline of *superscript* up; the default is for the rendering agent to use its own positioning rules.	

3.4.3 Subscript-superscript Pair `<msubsup>`[core]

3.4.3.1 Description

The `msubsup` element is used to attach both a subscript and superscript to a base expression.

```
<msubsup> base subscript superscript </msubsup>
```

It increments `scriptlevel` by 1, and sets `displaystyle` to `false`, within *subscript* and *superscript*, but leaves both attributes unchanged within *base*. (See 3.1.6 Displaystyle and Scriptlevel.)

Note that both scripts are positioned tight against the base as shown here x_1^2 versus the staggered positioning of nested scripts as shown here ${x_1}^2$; the latter can be achieved by nesting an `msub` inside an `msup`.

3.4.3.2 Attributes

`msubsup` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
subscriptshift[not-core]	*length*	*automatic*
	Specifies the minimum amount to shift the baseline of *subscript* down; the default is for the rendering agent to use its own positioning rules.	
superscriptshift[not-core]	*length*	*automatic*
	Specifies the minimum amount to shift the baseline of *superscript* up; the default is for the rendering agent to use its own positioning rules.	

3.4.3.3 Examples

▸ Show Section

The `msubsup` is most commonly used for adding sub-/superscript pairs to identifiers as illustrated above. However, another important use is placing limits on certain large operators whose limits are traditionally displayed in the script positions even when rendered in display style. The most common of these is the integral. For example,

$$\int_0^1 \mathrm{e}^x \,\mathrm{d}x$$

would be represented as

```
<mrow>
  <msubsup>
    <mo> ∫ </mo>
    <mn> 0 </mn>
    <mn> 1 </mn>
  </msubsup>
  <mrow>
    <msup>
      <mi> ⅇ </mi>
      <mi> x </mi>
    </msup>
    <mo> &#x2062;<!--InvisibleTimes--> </mo>
    <mrow>
      <mo> ⅆ </mo>
      <mi> x </mi>
    </mrow>
  </mrow>
</mrow>
```

$$\int_0^1 ⅇ^x ⅆx$$

3.4.4 Underscript `<munder>`[core]

3.4.4.1 Description

The `munder` element attaches an accent or limit placed under a base using the syntax

```
<munder> base underscript </munder>
```

It always sets `displaystyle` to `false` within the underscript, but increments `scriptlevel` by 1 only when `accentunder` is `false`. Within *base*, it always leaves both attributes unchanged. (See 3.1.6 Displaystyle and Scriptlevel.)

If *base* is an operator with `movablelimits=true` (or an embellished operator whose `mo` element core has `movablelimits=true`), and `displaystyle=false`, then *underscript* is drawn in a subscript position. In this case, the `accentunder` attribute is ignored. This is often used for limits on symbols such as U+2211 (entity `sum`).

3.4.4.2 Attributes

`munder` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
accentunder[core]	"true" | "false"	*automatic*
	Specifies whether *underscript* is drawn as an "accent" or as a limit. An accent is drawn the same size as the base (without incrementing `scriptlevel`) and is drawn closer to the base.	
align[not-core]	"left" | "right" | "center"	center
	Specifies whether the script is aligned left, center, or right under/over the base. As specified in 3.2.5.7.3 Horizontal Stretching Rules, the core of underscripts that are embellished operators should stretch to cover the base, but the alignment is based on the entire underscript.	

The default value of `accentunder` is false, unless *underscript* is an `mo` element or an embellished operator (see 3.2.5 Operator, Fence, Separator or Accent <mo>). If *underscript* is an `mo` element, the value of its `accent` attribute is used as the default value of `accentunder`. If *underscript* is an embellished operator, the `accent` attribute of the `mo` element at its core is used as the default value. As with all attributes, an explicitly given value overrides the default.

[MathML-Core] does not support the `accent` attribute on 3.2.5 Operator, Fence, Separator or Accent <mo>. For compatibility with MathML Core, the `accentunder` should be set on `munder`.

3.4.4.3 Examples

▸ Show Section

An example demonstrating how `accentunder` affects rendering:

```
<mrow>
  <munder accentunder="true">
    <mrow>
      <mi> x </mi>
      <mo> + </mo>
      <mi> y </mi>
      <mo> + </mo>
      <mi> z </mi>
    </mrow>
    <mo> ⏟ </mo>
  </munder>
  <mtext> <!--nbsp-->versus <!--nbsp--></mtext>
  <munder accentunder="false">
    <mrow>
      <mi> x </mi>
      <mo> + </mo>
      <mi> y </mi>
      <mo> + </mo>
      <mi> z </mi>
    </mrow>
    <mo> ⏟ </mo>
  </munder>
</mrow>
```

$$\underbrace{x + y + z}\ \text{versus}\ \underbrace{x + y + z}$$

3.4.5 Overscript `<mover>`[core]

3.4.5.1 Description

The `mover` element attaches an accent or limit placed over a base using the syntax

```
<mover> base overscript </mover>
```

It always sets `displaystyle` to `false` within overscript, but increments `scriptlevel` by 1 only when `accent` is `false`. Within *base*, it always leaves both attributes unchanged. (See 3.1.6 Displaystyle and Scriptlevel.)

If *base* is an operator with `movablelimits=true` (or an embellished operator whose `mo` element core has `movablelimits=true`), and `displaystyle=false`, then *overscript* is drawn in a superscript position. In this case, the `accent` attribute is ignored. This is often used for limits on symbols such as U+2211 (entity `sum`).

3.4.5.2 Attributes

`mover` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
accent[core]	"true" | "false"	*automatic*
	Specifies whether *overscript* is drawn as an "accent" or as a limit. An accent is drawn the same size as the base (without incrementing `scriptlevel`) and is drawn closer to the base.	
align[not-core]	"left" | "right" | "center"	center
	Specifies whether the script is aligned left, center, or right under/over the base. As specified in 3.2.5.7.3 Horizontal Stretching Rules, the core of overscripts that are embellished operators should stretch to cover the base, but the alignment is based on the entire overscript.	

The difference between an accent versus limit is shown in the examples.

The default value of *accent* is false, unless *overscript* is an `mo` element or an embellished operator (see 3.2.5 Operator, Fence, Separator or Accent <mo>). If *overscript* is an `mo` element, the value of its `accent` attribute is used as the default value of `accent` for `mover`. If *overscript* is an embellished operator, the `accent` attribute of the `mo` element at its core is used as the default value.

[MathML-Core] does not support the `accent` attribute on 3.2.5 Operator, Fence, Separator or Accent <mo>. For compatibility with MathML Core, the `accentunder` should be set on `munder`.

3.4.5.3 Examples

▶ Show Section

Two examples demonstrating how `accent` affects rendering:

```
<mrow>
  <mover accent="true">
    <mi> x </mi>
    <mo> ^ </mo>
  </mover>
  <mtext> <!--nbsp-->versus <!--nbsp--></mtext>
  <mover accent="false">
    <mi> x </mi>
    <mo> ^ </mo>
  </mover>
</mrow>
```

$\hat{x}$ versus $\hat{x}$

```
<mrow>
  <mover accent="true">
    <mrow>
      <mi> x </mi>
      <mo> + </mo>
      <mi> y </mi>
      <mo> + </mo>
      <mi> z </mi>
    </mrow>
    <mo> ‾ </mo>
  </mover>
  <mtext> <!--nbsp-->versus <!--nbsp--></mtext>
  <mover accent="false">
    <mrow>
      <mi> x </mi>
      <mo> + </mo>
      <mi> y </mi>
      <mo> + </mo>
      <mi> z </mi>
    </mrow>
    <mo> ‾ </mo>
  </mover>
</mrow>
```

$\overbrace{x + y + z}$ versus $\overbrace{x + y + z}$

3.4.6 Underscript-overscript Pair <munderover>[core]

3.4.6.1 Description

The `munderover` element attaches accents or limits placed both over and under a base using the syntax

```
<munderover> base underscript overscript </munderover>
```

It always sets `displaystyle` to `false` within *underscript* and *overscript*, but increments `scriptlevel` by 1 only when `accentunder` or `accent`, respectively, are `false`. Within *base*, it always leaves both attributes unchanged. (see 3.1.6 Displaystyle and Scriptlevel).

If *base* is an operator with `movablelimits=true` (or an embellished operator whose `mo` element core has `movablelimits=true`), and `displaystyle=false`, then *underscript* and *overscript* are drawn in a subscript and superscript position, respectively. In this case, the `accentunder` and `accent` attributes are ignored. This is often used for limits on symbols such as U+2211 (entity `sum`).

3.4.6.2 Attributes

`munderover` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
accent[core]	"true" | "false"	*automatic*
	Specifies whether *overscript* is drawn as an "accent" or as a limit. An accent is drawn the same size as the base (without incrementing `scriptlevel`) and is drawn closer to the base.	
accentunder[core]	"true" | "false"	*automatic*
	Specifies whether *underscript* is drawn as an "accent" or as a limit. An accent is drawn the same size as the base (without incrementing `scriptlevel`) and is drawn closer to the base.	
align[not-core]	"left" | "right" | "center"	center
	Specifies whether the scripts are aligned left, center, or right under/over the base. As specified in 3.2.5.7.3 Horizontal Stretching Rules, the core of underscripts and overscripts that are embellished operators should stretch to cover the base, but the alignment is based on the entire underscript or overscript.	

The `munderover` element is used instead of separate `munder` and `mover` elements so that the underscript and overscript are vertically spaced equally in relation to the base and so that they follow the slant of the base as shown in the example.

The defaults for `accent` and `accentunder` are computed in the same way as for `munder` and `mover`, respectively.

3.4.6.3 Examples

▸ Show Section

This example shows the difference between nesting `munder` inside `mover` and using `munderover` when `movablelimits=true` and in `displaystyle` (which renders the same as `msubsup`).

```
<mstyle displaystyle="false">
  <mover>
    <munder>
      <mo>∑</mo>
      <mi>i</mi>
    </munder>
    <mi>n</mi>
  </mover>
  <mo>+</mo>
  <munderover>
    <mo>∑</mo>
    <mi>i</mi>
    <mi>n</mi>
  </munderover>
</mstyle>
```

$$\sum_{i}^{n} + \sum_{i}^{n}$$

3.4.7 Prescripts and Tensor Indices <mmultiscripts>core, <mprescripts/>core

3.4.7.1 Description

Presubscripts and tensor notations are represented by a single element, `mmultiscripts`, using the syntax:

```
<mmultiscripts>
 base
 (subscript superscript)*
 [ <mprescripts/> (presubscript presuperscript)* ]
</mmultiscripts>
```

This element allows the representation of any number of vertically-aligned pairs of subscripts and superscripts, attached to one base expression. It supports both postscripts and prescripts. Missing scripts must be represented by a valid empty element denoting the empty subterm, such as `<mrow/>`. (The element `<none/>` was used in earlier MathML releases, but was equivalent to an empty `<mrow/>`). All of the upper scripts should be baseline-aligned and all the lower scripts should be baseline-aligned.

The prescripts are optional, and when present are given *after* the postscripts. This order was chosen because prescripts are relatively rare compared to tensor notation.

The argument sequence consists of the base followed by zero or more pairs of vertically-aligned subscripts and superscripts (in that order) that represent all of the postscripts. This list is optionally followed by an empty element `mprescripts` and a list of zero or more pairs of vertically-aligned presubscripts and presuperscripts that represent all of the prescripts. The pair lists for postscripts and prescripts are displayed in the same order as the directional context (i.e. left-to-right order in LTR context). If no subscript or superscript should be rendered in a given position, then an empty element `<mrow/>` should be used in that position. For each sub- and superscript pair, horizontal-alignment of the elements in the pair should be towards the base of the `mmultiscripts`. That is, pre-scripts should be right aligned, and post-scripts should be left aligned.

The base, subscripts, superscripts, the optional separator element `mprescripts`, the presubscripts, and the presuperscripts are all direct sub-expressions of the `mmultiscripts` element, i.e. they are all at the same level of the expression tree. Whether a script argument is a subscript or a superscript, or whether it is a presubscript or a presuperscript is determined by whether it occurs in an even-numbered or odd-numbered argument position, respectively, ignoring the empty element `mprescripts` itself when determining the position. The first argument, the base, is considered to be in position 1. The total number of arguments must be odd, if `mprescripts` is not given, or even, if it is.

The empty element `mprescripts` is only allowed as direct sub-expression of `mmultiscripts`.

3.4.7.2 Attributes

Same as the attributes of `msubsup`. See 3.4.3.2 Attributes.

The `mmultiscripts` element increments `scriptlevel` by 1, and sets `displaystyle` to `false`, within each of its arguments except *base*, but leaves both attributes unchanged within *base*. (See 3.1.6 Displaystyle and Scriptlevel.)

3.4.7.3 Examples

▸ Show Section

This example of a hypergeometric function demonstrates the use of pre and post subscripts:

```
<mrow>
  <mmultiscripts>
    <mi> F </mi>
    <mn> 1 </mn>
    <mrow/>
    <mprescripts/>
    <mn> 0 </mn>
    <mrow/>
  </mmultiscripts>
  <mo> &#x2061;<!--ApplyFunction--> </mo>
  <mrow>
    <mo> ( </mo>
    <mrow>
      <mo> ; </mo>
      <mi> a </mi>
      <mo> ; </mo>
      <mi> z </mi>
    </mrow>
    <mo> ) </mo>
  </mrow>
</mrow>
```

$$ {}_0F_1(;a;z) $$

This example shows a tensor. In the example, k and l are different indices

```
<mmultiscripts>
  <mi> R </mi>
  <mi> i </mi>
  <mrow/>
  <mrow/>
  <mi> j </mi>
  <mi> k </mi>
  <mrow/>
  <mi> l </mi>
  <mrow/>
</mmultiscripts>
```

$$ R_i{}^j{}_{kl} $$

This example demonstrates alignment towards the base of the scripts:

```
<mmultiscripts>
  <mi>  X </mi>
  <mn> 123 </mn>
  <mn> 1 </mn>
  <mprescripts/>
  <mn> 123 </mn>
  <mn> 1 </mn>
</mmultiscripts>
```

This final example of `mmultiscripts` shows how the binomial coefficient can be displayed in Arabic style

```
<mstyle dir="rtl">
  <mmultiscripts><mo>&#x0644;</mo>
    <mn>12</mn><mrow/>
    <mprescripts/>
    <mrow/><mn>5</mn>
  </mmultiscripts>
</mstyle>
```

3.5 Tabular Math

Matrices, arrays and other table-like mathematical notation are marked up using `mtable`, `mtr` and `mtd` elements. These elements are similar to the `table`, `tr` and `td` elements of HTML, except that they provide specialized attributes for the fine layout control necessary for commutative diagrams, block matrices and so on.

While the two-dimensional layouts used for elementary math such as addition and multiplication are somewhat similar to tables, they differ in important ways. For layout and for accessibility reasons, the `mstack` and `mlongdiv` elements discussed in 3.6 Elementary Math should be used for elementary math notations.

3.5.1 Table or Matrix `<mtable>` core

3.5.1.1 Description

A matrix or table is specified using the `mtable` element. Inside of the `mtable` element, only `mtr` elements may appear.

Table rows that have fewer columns than other rows of the same table (whether the other rows precede or follow them) are

effectively padded on the right (or left in RTL context) with empty `mtd` elements so that the number of columns in each row equals the maximum number of columns in any row of the table. Note that the use of `mtd` elements with non-default values of the `rowspan` or `columnspan` attributes may affect the number of `mtd` elements that should be given in subsequent `mtr` elements to cover a given number of columns.

MathML does not specify a table layout algorithm. In particular, it is the responsibility of a MathML renderer to resolve conflicts between the `width` attribute and other constraints on the width of a table, such as explicit values for `columnwidth` attributes, and minimum sizes for table cell contents. For a discussion of table layout algorithms, see Cascading Style Sheets, level 2.

§ *3.5.1.2 Attributes*

`mtable` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements. Any rules drawn as part of the `mtable` should be drawn using the color specified by `mathcolor`.

Name	values	default
align[not-core]	("top" | "bottom" | "center" | "baseline" | "axis"), *rownumber*?	axis
	specifies the vertical alignment of the table with respect to its environment. `axis` means to align the vertical center of the table on the environment's axis. (The ***axis*** of an equation is an alignment line used by typesetters. It is the line on which a minus sign typically lies.) `center` and `baseline` both mean to align the center of the table on the environment's baseline. `top` or `bottom` aligns the top or bottom of the table on the environment's baseline. If the `align` attribute value ends with a *rownumber*, the specified row (counting from 1 for the top row), rather than the table as a whole, is aligned in the way described above with the exceptions noted below. If *rownumber* is negative, it counts rows from the bottom. When the value of *rownumber* is out of range or not an integer, it is ignored. If a row number is specified and the alignment value is `baseline` or `axis`, the row's baseline or axis is used for alignment. Note this is only well defined when the `rowalign` value is `baseline` or `axis`; MathML does not specify how `baseline` or `axis` alignment should occur for other values of `rowalign`.	
rowalign[not-core]	("top" | "bottom" | "center" | "baseline" | "axis") +	baseline
	specifies the vertical alignment of the cells with respect to other cells within the same row: `top` aligns the tops of each entry across the row; `bottom` aligns the bottoms of the cells, `center` centers the cells; `baseline` aligns the baselines of the cells; `axis` aligns the axis of each cells. (See the note below about multiple values.)	
columnalign[not-core]	("left" | "center" | "right") +	center
	specifies the horizontal alignment of the cells with respect to other cells within the same column: `left` aligns the left side of the cells; `center` centers each cells; `right` aligns the right side of the cells. (See the note below about multiple values.)	
alignmentscope[not-core]	("true" | "false") +	true
	[this attribute is described with the alignment elements, `maligngroup` and `malignmark`, in 3.5.4 Alignment Markers <maligngroup/>, <malignmark/>[not-core].]	
columnwidth[not-core]	("auto" | *length* | "fit") +	auto
	specifies how wide a column should be: `auto` means that the column should be as wide as needed; an explicit length means that the column is exactly that wide and the contents of that column are made to fit by linewrapping or clipping at the discretion of the renderer; `fit` means that the page width remaining after subtracting the `auto` or fixed width columns is divided equally among the `fit` columns. If insufficient room remains to hold the contents of the `fit` columns, renderers may linewrap or clip the contents of the `fit` columns. Note that when the `columnwidth` is specified as a percentage, the value is relative to the width of the table, not as a percentage of the default (which is `auto`). That is, a renderer should try to adjust the width of the column so that it covers the specified percentage of the entire table width. (See the note below about multiple values.)	
width[not-core]	"auto" | *length*	auto
	specifies the desired width of the entire table and is intended for visual user agents. When the	

	specifies the desired width of the entire table and is intended for visual user agents. When the value is a percentage value, the value is relative to the horizontal space that a MathML renderer has available, this is the current target width as used for linebreaking as specified in 3.1.7 Linebreaking of Expressions; this allows the author to specify, for example, a table being full width of the display. When the value is `auto`, the MathML renderer should calculate the table width from its contents using whatever layout algorithm it chooses. Note: numbers without units were allowed in MathML 3 and treated similarly to percentage values, but unitless numbers are deprecated in MathML 4.	
rowspacingnot-core	(*length*) +	1.0ex
	specifies how much space to add between rows. (See the note below about multiple values.)	
columnspacingnot-core	(*length*) +	0.8em
	specifies how much space to add between columns. (See the note below about multiple values.)	
rowlinesnot-core	("none" | "solid" | "dashed") +	none
	specifies whether and what kind of lines should be added between each row: `none` means no lines; `solid` means solid lines; `dashed` means dashed lines (how the dashes are spaced is implementation dependent). (See the note below about multiple values.)	
columnlinesnot-core	("none" | "solid" | "dashed") +	none
	specifies whether and what kind of lines should be added between each column: `none` means no lines; `solid` means solid lines; `dashed` means dashed lines (how the dashes are spaced is implementation dependent). (See the note below about multiple values.)	
framenot-core	"none" | "solid" | "dashed"	none
	specifies whether and what kind of lines should be drawn around the table. `none` means no lines; `solid` means solid lines; `dashed` means dashed lines (how the dashes are spaced is implementation dependent).	
framespacingnot-core	*length*, *length*	0.4em 0.5ex
	specifies the additional spacing added between the table and frame, if `frame` is not `none`. The first value specifies the spacing on the right and left; the second value specifies the spacing above and below.	
equalrowsnot-core	"true" | "false"	false
	specifies whether to force all rows to have the same total height.	
equalcolumnsnot-core	"true" | "false"	false
	specifies whether to force all columns to have the same total width.	
displaystylenot-core	"true" | "false"	false
	specifies the value of `displaystyle` within each cell (`scriptlevel` is not changed); see 3.1.6 Displaystyle and Scriptlevel.	

In the above specifications for attributes affecting rows (respectively, columns, or the gaps between rows or columns), the notation `(...)+` means that multiple values can be given for the attribute as a space separated list (see 2.1.5 MathML Attribute Values). In this context, a single value specifies the value to be used for all rows (resp., columns or gaps). A list of values are taken to apply to corresponding rows (resp., columns or gaps) in order, that is starting from the top row for rows or first column (left or right, depending on directionality) for columns. If there are more rows (resp., columns or gaps) than supplied values, the last value is repeated as needed. If there are too many values supplied, the excess are ignored.

Note that none of the areas occupied by lines `frame`, `rowlines` and `columnlines`, nor the spacing `framespacing`, `rowspacing` or `columnspacing` are counted as rows or columns.

The `displaystyle` attribute is allowed on the `mtable` element to set the inherited value of the attribute. If the attribute is not present, the `mtable` element sets `displaystyle` to `false` within the table elements. (See 3.1.6 Displaystyle and Scriptlevel.)

3.5.1.3 Examples

▸ Show Section

A 3 by 3 identity matrix could be represented as follows:

```
<mrow>
  <mo> ( </mo>
  <mtable>
    <mtr>
      <mtd> <mn>1</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
    </mtr>
    <mtr>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>1</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
    </mtr>
    <mtr>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>0</mn> </mtd>
      <mtd> <mn>1</mn> </mtd>
    </mtr>
  </mtable>
  <mo> ) </mo>
</mrow>
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

> Note that the parentheses must be represented explicitly; they are not part of the `mtable` element's rendering. This allows use of other surrounding fences, such as brackets, or none at all.

3.5.2 Row in Table or Matrix `<mtr>` core

3.5.2.1 Description

An `mtr` element represents one row in a table or matrix. An `mtr` element is only allowed as a direct sub-expression of an `mtable` element, and specifies that its contents should form one row of the table. Each argument of `mtr` is placed in a different column of the table, starting at the leftmost column in a LTR context or rightmost column in a RTL context.

As described in 3.5.1 Table or Matrix `<mtable>`, `mtr` elements are effectively padded with `mtd` elements when they are shorter than other rows in a table.

3.5.2.2 Attributes

`mtr` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
rowalign not-core	"top" | "bottom" | "center" | "baseline" | "axis"	*inherited*
	overrides, for this row, the vertical alignment of cells specified by the `rowalign` attribute on the `mtable`.	
columnalign not-core	("left" | "center" | "right") +	*inherited*
	overrides, for this row, the horizontal alignment of cells specified by the `columnalign` attribute on the `mtable`.	

3.5.2.3 Equation Numbering

Earlier versions of MathML specified an `mlabeledtr` element for numbered equations. In an `mlabeledtr`, the first `mtd` represents the equation number and the remaining elements in the row the equation being numbered. The `side` and `minlabelspacing` attributes of `mtable` determines the placement of the equation number. This element was not widely implemented and is not specified in the current version, it is still valid in the Legacy Schema.

In larger documents with many numbered equations, automatic numbering becomes important. While automatic equation numbering and automatically resolving references to equation numbers is outside the scope of MathML, these problems can

be addressed by the use of style sheets or other means. In a CSS context, one could use an empty `mtd` as the first child of a `mtr` and use CSS counters and generated content to fill in the equation number using a CSS style such as

```
body {counter-reset: eqnum;}
mtd.eqnum {counter-increment: eqnum;}
mtd.eqnum:before {content: "(" counter(eqnum) ")"}
```

3.5.3 Entry in Table or Matrix <mtd>core

3.5.3.1 Description

An `mtd` element represents one entry, or cell, in a table or matrix. An `mtd` element is only allowed as a direct sub-expression of an `mtr` element.

The `mtd` element accepts a single argument possibly being an inferred `mrow` of multiple children; see 3.1.3 Required Arguments.

3.5.3.2 Attributes

`mtd` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
rowspan[core]	*positive-integer*	1
	causes the cell to be treated as if it occupied the number of rows specified. The corresponding `mtd` in the following `rowspan`-1 rows must be omitted. The interpretation corresponds with the similar attributes for HTML tables.	
columnspan[core]	*positive-integer*	1
	causes the cell to be treated as if it occupied the number of columns specified. The following `rowspan`-1 `mtd`s must be omitted. The interpretation corresponds with the similar attributes for HTML tables.	
rowalign[not-core]	"top" | "bottom" | "center" | "baseline" | "axis"	*inherited*
	specifies the vertical alignment of this cell, overriding any value specified on the containing `mrow` and `mtable`. See the `rowalign` attribute of `mtable`.	
columnalign[not-core]	"left" | "center" | "right"	*inherited*
	specifies the horizontal alignment of this cell, overriding any value specified on the containing `mrow` and `mtable`. See the `columnalign` attribute of `mtable`.	

3.5.4 Alignment Markers `<maligngroup/>`, `<malignmark/>`[not-core]

> NOTE: malignmark and maligngroup are not in MathML-Core
>
> `malignmark` and `maligngroup` are not supported in [MathML-Core]. For most purposes it is recommended that alignment is implemented directly using `mtable` columns. As noted in the following section these elements may be futher simplified or removed in a future version of MathML.
>
> For existing MathML using `malignmark` a Javascript polyfill is provided.

3.5.4.1 Removal Notice

With one significant exception, `<maligngroup/>` and `<malignmark/>` have had minimal adoption and implementation. The one exception only uses the basics of alignment. Because of this, alignment in MathML is significantly simplified to align with the current usage and make future implementation simplier. In particular, the following simplifications are made:

- the attributes for `<maligngroup/>` and `<malignmark/>` have been removed.
- The `groupalign` attribute previously allowed on `mtable`, `mtr`, and `mlabeledtr` is removed
- `<malignmark/>` used to be allowed anywhere, including inside of token elements; it is now allowed in only the

locations that `<maligngroup/>` is allowed (see below)

3.5.4.2 Description

Alignment markers are space-like elements (see 3.2.7 Space `<mspace/>`) that can be used to vertically align specified points within a column of MathML expressions by the automatic insertion of the necessary amount of horizontal space between specified sub-expressions.

The discussion that follows will use the example of a set of simultaneous equations that should be rendered with vertical alignment of the coefficients and variables of each term, by inserting spacing somewhat like that shown here:

$$8.44x + 55\ \ y = \ \ 0$$
$$3.1\ \ x - \ \ 0.7y = -1.1$$

If the example expressions shown above were arranged in a column but not aligned, they would appear as:

$$8.44x + 55.7y = 0$$
$$3.1x - 50.7y = -1.1$$

The expressions whose parts are to be aligned (each equation, in the example above) must be given as the table elements (i.e. as the `mtd` elements) of one column of an `mtable`. To avoid confusion, the term “table cell” rather than “table element” will be used in the remainder of this section.

All interactions between alignment elements are limited to the `mtable` column they arise in. That is, every column of a table specified by an `mtable` element acts as an “alignment scope” that contains within it all alignment effects arising from its contents. It also excludes any interaction between its own alignment elements and the alignment elements inside any nested alignment scopes it might contain.

If there is only one alignment point, an alternative is to use linebreaking and indentation attributes on `mo` elements as described in 3.1.7 Linebreaking of Expressions.

An `mtable` element can be given the attribute `alignmentscope=false` to cause its columns not to act as alignment scopes. This is discussed further at the end of this section. Otherwise, the discussion in this section assumes that this attribute has its default value of `true`.

3.5.4.3 Specifying alignment groups

Each part of expression to be aligned should be in an `maligngroup`. The point of alignment is the left edge (right edge if for RTL) of the element that follows an `maligngroup` element unless an `malignmark` element is between `maligngroup` elements. In that case, the left edge (right edge if for RTL) of the element that follows the `malignmark` is the point of alignment for that group.

If `maligngroup` or `maligngroup` occurs outside of an `mtable`, they are rendered with zero width.

In the example above, each equation would have one `maligngroup` element before each coefficient, variable, and operator on the left-hand side, one before the = sign, and one before the constant on the right-hand side because these are the parts that should be aligned.

In general, a table cell containing *n* `maligngroup` elements contains *n* alignment groups, with the *i*th group consisting of the elements entirely after the *i*th `maligngroup` element and before the (*i*+1)-th; no element within the table cell's content should occur entirely before its first `maligngroup` element.

Note that the division into alignment groups does *not* necessarily fit the nested expression structure of the MathML expression containing the groups — that is, it is permissible for one alignment group to consist of the end of one `mrow`, all of another one, and the beginning of a third one, for example. This can be seen in the MathML markup for the example given at the end of this section.

Although alignment groups need not coincide with the nested expression structure of layout schemata, there are nonetheless restrictions on where `maligngroup` and `malignmark` elements are allowed within a table cell. These elements may only be contained within elements (directly or indirectly) of the following types (which are themselves contained in the table cell):

- an `mrow` element, including an inferred `mrow` such as the one formed by a multi-child `mtd` element, but excluding `mrow` which contains a change of direction using the `dir` attribute;
- an `mstyle` element , but excluding those which change direction using the `dir` attribute;
- an `mphantom` element;
- an `mfenced` element;
- an `maction` element, though only its selected sub-expression is checked;
- a `semantics` element.

These restrictions are intended to ensure that alignment can be unambiguously specified, while avoiding complexities involving things like overscripts, radical signs and fraction bars. They also ensure that a simple algorithm suffices to accomplish the desired alignment.

For the table cells that are divided into alignment groups, every element in their content must be part of exactly one alignment group, except for the elements from the above list that contain `maligngroup` elements inside them and the `maligngroup` elements themselves. This means that, within any table cell containing alignment groups, the first complete element must be an `maligngroup` element, though this may be preceded by the start tags of other elements. This requirement removes a potential confusion about how to align elements before the first `maligngroup` element, and makes it easy to identify table cells that are left out of their column's alignment process entirely.

It is not required that the table cells in a column that are divided into alignment groups each contain the same number of groups. If they don't, zero-width alignment groups are effectively added on the right side (or left side, in a RTL context) of each table cell that has fewer groups than other table cells in the same column.

> NOTE
>
> Do we want to tighten this so that all rows have the same number of `maligngroup` elements?

3.5.4.4 Table cells that are not divided into alignment groups

> NOTE
>
> Do we still want to allow rows without `maligngroup` as described in this section?

Expressions in a column that are to have no alignment groups should contain no `maligngroup` elements. Expressions with no alignment groups are aligned using only the `columnalign` attribute that applies to the table column as a whole. If such an expression is wider than the column width needed for the table cells containing alignment groups, all the table cells containing alignment groups will be shifted as a unit within the column as described by the `columnalign` attribute for that column. For example, a column heading with no internal alignment could be added to the column of two equations given above by preceding them with another table row containing an `mtext` element for the heading, and using the default `columnalign`="center" for the table, to produce:

> equations with aligned variables
>
> $8.44x + 55 y = 0$
>
> $3.1 x - 0.7y = -1.1$

or, with a shorter heading,

> some equations
>
> $8.44x + 55 y = 0$
>
> $3.1 x - 0.7y = -1.1$

3.5.4.5 Specifying alignment points using `<malignmark/>`

An `malignmark` element anywhere within the alignment group (except within another alignment scope wholly contained inside it) overrides alignment at the start of an `maligngroup` element.

The `malignmark` element indicates that the alignment point should occur on the left edge (right edge in a RTL context) of the following element.

> NOTE
>
> Can `malignmark` elements occur inside of tokens?

When an `malignmark` element is provided within an alignment group, it should only occur within the elements allowed for `maligngroup` (see 3.5.4.3 Specifying alignment groups). If there is more than one `malignmark` element in an alignment group, all but the first one will be ignored. MathML applications may wish to provide a mode in which they will warn about this situation, but it is not an error, and should trigger no warnings by default. The rationale for this is that it would be inconvenient to have to remove all unnecessary `malignmark` elements from automatically generated data.

3.5.4.6 MathML representation of an alignment example

▶ Show Section

The above rules are sufficient to explain the MathML representation of the example given near the start of this section.

ISSUE 180: "decimalpoint" value definition **MathML 4** **compatibility** **need specification update**

issue 180

One way to represent that in MathML is:

```
<mtable groupalign="{decimalpoint left left decimalpoint left left decimalpoint}">
  <mtr>
    <mtd>
      <mrow>
        <mrow>
          <mrow>
            <maligngroup/>
            <mn> 8.44 </mn>
            <mo> &#x2062;<!--InvisibleTimes--> </mo>
            <maligngroup/>
            <mi> x </mi>
          </mrow>
          <maligngroup/>
          <mo> + </mo>
          <mrow>
            <maligngroup/>
            <mn> 55 </mn>
            <mo> &#x2062;<!--InvisibleTimes--> </mo>
            <maligngroup/>
            <mi> y </mi>
          </mrow>
        </mrow>
        <maligngroup/>
        <mo> = </mo>
        <maligngroup/>
        <mn> 0 </mn>
      </mrow>
    </mtd>
    </mtr>
    <mtr>
      <mtd>
        <mrow>
          <mrow>
            <mrow>
              <maligngroup/>
              <mn> 3.1 </mn>
              <mo> &#x2062;<!--InvisibleTimes--> </mo>
              <maligngroup/>
              <mi> x </mi>
            </mrow>
            <maligngroup/>
            <mo> − </mo>
            <mrow>
              <maligngroup/>
              <mn> 0.7 </mn>
              <mo> &#x2062;<!--InvisibleTimes--> </mo>
              <maligngroup/>
              <mi> y </mi>
            </mrow>
          </mrow>
          <maligngroup/>
          <mo> = </mo>
```

```
          <maligngroup/>
          <mrow>
            <mo> - </mo>
            <mn> 1.1 </mn>
          </mrow>
        </mrow>
      </mtd>
    </mtr>
  </mtable>
```

$$\begin{aligned} 8.44x + 55 \;\; y &= \;\; 0 \\ 3.1 \;\; x - \;\; 0.7y &= -1.1 \end{aligned}$$

3.5.4.7 A simple alignment algorithm

A simple algorithm by which a MathML renderer can perform the alignment specified in this section is given here. Since the alignment specification is deterministic (except for the definition of the left and right edges of a character), any correct MathML alignment algorithm will have the same behavior as this one. Each `mtable` column (alignment scope) can be treated independently; the algorithm given here applies to one `mtable` column, and takes into account the alignment elements and the `columnalign` attribute described under `mtable` (3.5.1 Table or Matrix <mtable>). In an RTL context, switch left and right edges in the algorithm.

> NOTE
>
> This algorithm should be verified by an implementation.

1. A rendering is computed for the contents of each table cell in the column, using zero width for all `maligngroup` and `malignmark` elements. The final rendering will be identical except for horizontal shifts applied to each alignment group and/or table cell.
2. For each alignment group, the horizontal positions of the left edge, alignment point (if specified by `malignmark`, otherwise the left edge), and right edge are noted, allowing the width of the group on each side of the alignment point (left and right) to be determined. The sum of these two "side-widths", i.e. the sum of the widths to the left and right of the alignment point, will equal the width of the alignment group.
3. Each column of alignment groups is scanned. The *i*th scan covers the *i*th alignment group in each table cell containing any alignment groups. Table cells with no alignment groups, or with fewer than *i* alignment groups, are ignored. Each scan computes two maximums over the alignment groups scanned: the maximum width to the left of the alignment point, and the maximum width to the right of the alignment point, of any alignment group scanned.
4. The sum of all the maximum widths computed (two for each column of alignment groups) gives one total width, which will be the width of each table cell containing alignment groups. Call the maximum number of alignment groups in one cell n; each such cell is divided into $2n$ horizontally adjacent sections, called L(i) and R(i) for i from 1 to n, using the $2n$ maximum side-widths computed above; for each i, the width of all sections called L(i) is the maximum width of any cell's ith alignment group to the left of its alignment point, and the width of all sections called R(i) is the maximum width of any cell's ith alignment group to the right of its alignment point.

5. Each alignment group is then shifted horizontally as a block to a unique position that places: in the section called L(*i*) that part of the *i*th group to the left of its alignment point; in the section called R(*i*) that part of the *i*th group to the right of its alignment point. This results in the alignment point of each *i*th group being on the boundary between adjacent sections L(*i*) and R(*i*), so that all alignment points of *i*th groups have the same horizontal position.

The widths of the table cells that contain no alignment groups were computed as part of the initial rendering, and may be different for each cell, and different from the single width used for cells containing alignment groups. The maximum of all the cell widths (for both kinds of cells) gives the width of the table column as a whole.

The position of each cell in the column is determined by the applicable part of the value of the `columnalign` attribute of the innermost surrounding `mtable`, `mtr`, or `mtd` element that has an explicit value for it, as described in the sections on those elements. This may mean that the cells containing alignment groups will be shifted within their column, in addition to their alignment groups having been shifted within the cells as described above, but since each such cell has the same width, it will be shifted the same amount within the column, thus maintaining the vertical alignment of the alignment points of the corresponding alignment groups in each cell.

3.6 Elementary Math

Mathematics used in the lower grades such as two-dimensional addition, multiplication, and long division tends to be tabular in nature. However, the specific notations used varies among countries much more than for higher level math. Furthermore, elementary math often presents examples in some intermediate state and MathML must be able to capture these intermediate or intentionally missing partial forms. Indeed, these constructs represent memory aids or procedural guides, as much as they represent ‘mathematics’.

The elements used for basic alignments in elementary math are:

`mstack`
: align rows of digits and operators

`msgroup`
: groups rows with similar alignment

`msrow`
: groups digits and operators into a row

`msline`
: draws lines between rows of the stack

`mscarries`
: annotates the following row with optional borrows/carries and/or crossouts

`mscarry`
: a borrow/carry and/or crossout for a single digit

`mlongdiv`
: specifies a divisor and a quotient for long division, along with a stack of the intermediate computations

`mstack` and `mlongdiv` are the parent elements for all elementary math layout. Any children of `mstack`, `mlongdiv`, and `msgroup`, besides `msrow`, `msgroup`, `mscarries` and `msline`, are treated as if implicitly surrounded by an `msrow` (see 3.6.4 Rows in Elementary Math <msrow>not-core for more details about rows).

Since the primary use of these stacking constructs is to stack rows of numbers aligned on their digits, and since numbers are always formatted left-to-right, the columns of an mstack are always processed left-to-right; the overall directionality in effect (i.e. the `dir` attribute) does not affect to the ordering of display of columns or carries in rows and, in particular, does not affect the ordering of any operators within a row (see 3.1.5 Directionality).

These elements are described in this section followed by examples of their use. In addition to two-dimensional addition, subtraction, multiplication, and long division, these elements can be used to represent several notations used for repeating decimals.

A very simple example of two-dimensional addition is shown below:

```
<mstack>
  <mn>424</mn>
  <msrow> <mo>+</mo> <mn>33</mn> </msrow>
  <msline/>
</mstack>
```

Many more examples are given in 3.6.8 Elementary Math Examples.

3.6.1 Stacks of Characters <mstack>[not-core]

3.6.1.1 Description

`mstack` is used to lay out rows of numbers that are aligned on each digit. This is common in many elementary math notations such as 2D addition, subtraction, and multiplication.

The children of an `mstack` represent rows, or groups of them, to be stacked each below the previous row; there can be any number of rows. An `msrow` represents a row; an `msgroup` groups a set of rows together so that their horizontal alignment can be adjusted together; an `mscarries` represents a set of carries to be applied to the following row; an `msline` represents a line separating rows. Any other element is treated as if implicitly surrounded by `msrow`.

Each row contains 'digits' that are placed into columns. (see 3.6.4 Rows in Elementary Math <msrow>[not-core] for further details). The `stackalign` attribute together with the `position` and `shift` attributes of `msgroup`, `mscarries`, and `msrow` determine to which column a character belongs.

The width of a column is the maximum of the widths of each 'digit' in that column — carries do *not* participate in the width calculation; they are treated as having zero width. If an element is too wide to fit into a column, it overflows into the adjacent column(s) as determined by the `charalign` attribute. If there is no character in a column, its width is taken to be the width of a 0 in the current language (in many fonts, all digits have the same width).

The method for laying out an mstack is:

1. The ‘digits’ in a row are determined.
2. All of the digits in a row are initially aligned according to the `stackalign` value.
3. Each row is positioned relative to that alignment based on the `position` attribute (if any) that controls that row.
4. The maximum width of the digits in a column are determined and shorter and wider entries in that column are aligned according to the `charalign` attribute.
5. The width and height of the mstack element are computed based on the rows and columns. Any overflow from a column is *not* used as part of that computation.
6. The baseline of the mstack element is determined by the `align` attribute.

3.6.1.2 Attributes

`mstack` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
align[not-core]	("top" | "bottom" | "center" | "baseline" | "axis"), *rownumber*?	baseline
	specifies the vertical alignment of the `mstack` with respect to its environment. The legal values and their meanings are the same as that for `mtable`'s `align` attribute.	
stackalign[not-core]	"left" | "center" | "right" | "decimalpoint"	decimalpoint
	specifies which column is used to horizontally align the rows. For `left`, rows are aligned flush on the left; similarly for `right`, rows are flush on the right; for `center`, the middle column (or to the right of the middle, for an even number of columns) is used for alignment. Rows with non-zero `position`, or affected by a `shift`, are treated as if the requisite number of empty columns were added on the appropriate side; see 3.6.3 Group Rows with Similar Positions `<msgroup>`[not-core] and 3.6.4 Rows in Elementary Math `<msrow>`[not-core]. For `decimalpoint`, the column used is the left-most column in each row that contains the decimalpoint character specified using the `decimalpoint` attribute of `mstyle` (default "."). If there is no decimalpoint character in the row, an implied decimal is assumed on the right of the first number in the row; see `decimalpoint` for a discussion of `decimalpoint`.	
charalign[not-core]	"left" | "center" | "right"	right
	specifies the horizontal alignment of digits within a column. If the content is larger than the column width, then it overflows the opposite side from the alignment. For example, for `right`, the content will overflow on the left side; for center, it overflows on both sides. This excess does not participate in the column width calculation, nor does it participate in the overall width of the `mstack`. In these cases, authors should take care to avoid collisions between column overflows.	
charspacing[not-core]	*length* | "loose" | "medium" | "tight"	medium
	specifies the amount of space to put between each column. Larger spacing might be useful if carries are not placed above or are particularly wide. The keywords `loose`, `medium`, and `tight` automatically adjust spacing to when carries or other entries in a column are wide. The three values allow authors to some flexibility in choosing what the layout looks like without having to figure out what values work well. In all cases, the spacing between columns is a fixed amount and does not vary between different columns.	

3.6.2 Long Division `<mlongdiv>`[not-core]

3.6.2.1 Description

Long division notation varies quite a bit around the world, although the heart of the notation is often similar. `mlongdiv` is similar to `mstack` and used to layout long division. The first two children of `mlongdiv` are the divisor and the result of the division, in that order. The remaining children are treated as if they were children of `mstack`. The placement of these and the lines and separators used to display long division are controlled by the `longdivstyle` attribute.

The result or divisor may be an elementary math element or may be an empty `<mrow/>` (the specific empty element `<none/>` used in MathML 3 is not used in this specification). In particular, if `msgroup` is used, the elements in that group may or may not form their own mstack or be part of the dividend's `mstack`, depending upon the value of the `longdivstyle` attribute. For example, in the US style for division, the result is treated as part of the dividend's `mstack`, but divisor is not. MathML does not specify when the result and divisor form their own `mstack`, nor does it specify what should happen if `msline` or other elementary math elements are used for the result or divisor and they do not participate in the dividend's `mstack` layout.

In the remainder of this section on elementary math, anything that is said about `mstack` applies to `mlongdiv` unless stated otherwise.

3.6.2.2 Attributes

`mlongdiv` elements accept all of the attributes that `mstack` elements accept (including those specified in 3.1.9 Mathematics attributes common to presentation elements), along with the attribute listed below.

The values allowed for `longdivstyle` are open-ended. Conforming renderers may ignore any value they do not handle, although renderers are encouraged to render as many of the values listed below as possible. Any rules drawn as part of division layout should be drawn using the color specified by `mathcolor`.

Name	values	default
longdivstyle[not-core]	"lefttop" | "stackedrightright" | "mediumstackedrightright" | "shortstackedrightright" | "righttop" | "left/\right" | "left)(right" | ":right=right" | "stackedleftleft" | "stackedleftlinetop"	lefttop
	Controls the style of the long division layout. The names are meant as a rough mnemonic that describes the position of the divisor and result in relation to the dividend.	

See 3.6.8.3 Long Division for examples of how these notations are drawn. The values listed above are used for long division notations in different countries around the world:

`lefttop`
: a notation that is commonly used in the United States, Great Britain, and elsewhere

`stackedrightright`
: a notation that is commonly used in France and elsewhere

`mediumrightright`
: a notation that is commonly used in Russia and elsewhere

`shortstackedrightright`
: a notation that is commonly used in Brazil and elsewhere

`righttop`
: a notation that is commonly used in China, Sweden, and elsewhere

`left/\right`

a notation that is commonly used in Netherlands

`left)(right`
a notation that is commonly used in India

`:right=right`
a notation that is commonly used in Germany

`stackedleftleft`
a notation that is commonly used in Arabic countries

`stackedleftlinetop`
a notation that is commonly used in Arabic countries

3.6.3 Group Rows with Similar Positions <msgroup>[not-core]

3.6.3.1 Description

`msgroup` is used to group rows inside of the `mstack` and `mlongdiv` elements that have a similar position relative to the alignment of stack. If not explicitly given, the children representing the stack in `mstack` and `mlongdiv` are treated as if they are implicitly surrounded by an `msgroup` element.

3.6.3.2 Attributes

`msgroup` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
position[not-core]	*integer*	0
	specifies the horizontal position of the rows within this group relative to the position determined by the containing `msgroup` (according to its `position` and `shift` attributes). The resulting position value is relative to the column specified by `stackalign` of the containing `mstack` or `mlongdiv`. Positive values move each row towards the tens digit, like multiplying by a power of 10, effectively padding with empty columns on the right; negative values move towards the ones digit, effectively padding on the left. The decimal point is counted as a column and should be taken into account for negative values.	
shift[not-core]	*integer*	0
	specifies an incremental shift of position for successive children (rows or groups) within this group. The value is interpreted as with position, but specifies the position of each child (except the first) with respect to the previous child in the group.	

3.6.4 Rows in Elementary Math `<msrow>`[not-core]

3.6.4.1 Description

An `msrow` represents a row in an `mstack`. In most cases it is implied by the context, but is useful explicitly for putting multiple elements in a single row, such as when placing an operator "+" or "-" alongside a number within an addition or subtraction.

If an `mn` element is a child of `msrow` (whether implicit or not), then the number is split into its digits and the digits are placed into successive columns. Any other element, with the exception of `mstyle` is treated effectively as a single digit occupying the next column. An `mstyle` is treated as if its children were directly the children of the `msrow`, but with their style affected by the attributes of the `mstyle`. The empty element `<mrow/>` may be used to create an empty column.

Note that a row is considered primarily as if it were a number, which is always displayed left-to-right, and so the directionality used to display the columns is always left-to-right; textual bidirectionality within token elements (other than `mn`) still applies, as does the overall directionality *within* any children of the `msrow` (which end up treated as single digits); see 3.1.5 Directionality.

3.6.4.2 Attributes

`msrow` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
position[not-core]	*integer*	0
	specifies the horizontal position of the rows within this group relative to the position determined by the containing `msgroup` (according to its `position` and `shift` attributes). The resulting position value is relative to the column specified by `stackalign` of the containing `mstack` or `mlongdiv`. Positive values move each row towards the tens digit, like multiplying by a power of 10, effectively padding with empty columns on the right; negative values move towards the ones digit, effectively padding on the left. The decimal point is counted as a column and should be taken into account for negative values.	

3.6.5 Carries, Borrows, and Crossouts <mscarries>[not-core]

3.6.5.1 Description

The `mscarries` element is used for various annotations such as carries, borrows, and crossouts that occur in elementary math. The children are associated with elements in the *following* row of the `mstack`. It is an error for `mscarries` to be the last element of an `mstack` or `mlongdiv` element. Each child of the `mscarries` applies to the same column in the following row. As these annotations are used to adorn what are treated as numbers, the attachment of carries to columns proceeds from left to right; the overall directionality does not apply to the ordering of the carries, although it may apply to the contents of each carry; see 3.1.5 Directionality.

Each child of `mscarries` other than `mscarry` or `<mrow/>` is treated as if implicitly surrounded by `mscarry`; the element `<mrow/>` is used when no carry for a particular column is needed. The element `<none/>` was used in earlier MathML releases, but was equivalent to an empty `<mrow/>`. The `mscarries` element sets `displaystyle` to `false`, and increments `scriptlevel` by 1, so the children are typically displayed in a smaller font. (See 3.1.6 Displaystyle and Scriptlevel.) It also changes the default value of `scriptsizemultiplier`. The effect is that the inherited value of `scriptsizemultiplier` should still override the default value, but the default value, inside `mscarries`, should be `0.6`. `scriptsizemultiplier` can be set on the `mscarries` element, and the value should override the inherited value as usual.

If two rows of carries are adjacent to each other, the first row of carries annotates the second (following) row as if the second row had `location=n`. This means that the second row, even if it does not draw, visually uses some (undefined by this specification) amount of space when displayed.

3.6.5.2 Attributes

`mscarries` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
position[not-core]	*integer*	0
	specifies the horizontal position of the rows within this group relative to the position determined by the containing `msgroup` (according to its `position` and `shift` attributes). The resulting position value is relative to the column specified by `stackalign` of the containing `mstack` or `mlongdiv`. The interpretation of the value is the same as `position` for `msgroup` or `msrow`, but it alters the association of each carry with the column below. For example, `position`=1 would cause the rightmost carry to be associated with the second digit column from the right.	
location[not-core]	"w" | "nw" | "n" | "ne" | "e" | "se" | "s" | "sw"	n
	specifies the location of the carry or borrow relative to the character below it in the associated column. Compass directions are used for the values; the default is to place the carry above the character.	
crossout[not-core]	("none" | "updiagonalstrike" | "downdiagonalstrike" | "verticalstrike" | "horizontalstrike")*	none
	specifies how the column content below each carry is "crossed out"; one or more values may be given and all values are drawn. If `none` is given with other values, it is ignored. See 3.6.8 Elementary Math Examples for examples of the different values. The crossout is only applied for columns which have a corresponding `mscarry`. The crossouts should be drawn using the color specified by `mathcolor`.	
scriptsizemultiplier[not-core]	*number*	*inherited (0.6)*
	specifies the factor to change the font size by. See 3.1.6 Displaystyle and Scriptlevel for a description of how this works with the `scriptsize` attribute.	

3.6.6 A Single Carry `<mscarry>`[not-core]

3.6.6.1 Description

`mscarry` is used inside of `mscarries` to represent the carry for an individual column. A carry is treated as if its width were zero; it does not participate in the calculation of the width of its corresponding column; as such, it may extend beyond the column boundaries. Although it is usually implied, the element may be used explicitly to override the `location` and/or `crossout` attributes of the containing `mscarries`. It may also be useful with `<mrow/>` as its content in order to display no actual carry, but still enable a `crossout` due to the enclosing `mscarries` to be drawn for the given column.

3.6.6.2 Attributes

The `mscarry` element accepts the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

Name	values	default
location[not-core]	"w" | "nw" | "n" | "ne" | "e" | "se" | "s" | "sw"	*inherited*
	specifies the location of the carry or borrow relative to the character in the corresponding column in the row below it. Compass directions are used for the values.	
crossout[not-core]	("none" | "updiagonalstrike" | "downdiagonalstrike" | "verticalstrike" | "horizontalstrike")*	*inherited*
	specifies how the column content associated with the carry is "crossed out"; one or more values may be given and all values are drawn. If `none` is given with other values, it is essentially ignored. The crossout should be drawn using the color specified by `mathcolor`.	

3.6.7 Horizontal Line `<msline/>`[not-core]

3.6.7.1 Description

`msline` draws a horizontal line inside of an `mstack` element. The position, length, and thickness of the line are specified as attributes. If the length is specified, the line is positioned and drawn as if it were a number with the given number of digits.

3.6.7.2 Attributes

`msline` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements. The line should be drawn using the color specified by `mathcolor`.

Name	values	default
position[not-core]	*integer*	0
	specifies the horizontal position of the rows within this group relative to the position determined by the containing `msgroup` (according to its `position` and `shift` attributes). The resulting position value is relative to the column specified by `stackalign` of the containing `mstack` or `mlongdiv`. Positive values move towards the tens digit (like multiplying by a power of 10); negative values move towards the ones digit. The decimal point is counted as a column and should be taken into account for negative values. Note that since the default line length spans the entire `mstack`, the position has no effect unless the `length` is specified as non-zero.	
length[not-core]	*unsigned-integer*	0
	Specifies the number of columns that should be spanned by the line. A value of '0' (the default) means that *all* columns in the row are spanned (in which case `position` and `stackalign` have no effect).	
leftoverhang[not-core]	*length*	0
	Specifies an extra amount that the line should overhang on the left of the leftmost column spanned by the line.	
rightoverhang[not-core]	*length*	0
	Specifies an extra amount that the line should overhang on the right of the rightmost column spanned by the line.	
mslinethickness[not-core]	*length* | "thin" | "medium" | "thick"	medium
	Specifies how thick the line should be drawn. The line should have height=0, and depth=mslinethickness so that the top of the `msline` is on the baseline of the surrounding context (if any). (See 3.3.2 Fractions `<mfrac>` for discussion of the thickness keywords `medium`, `thin` and `thick`.)	

3.6.8 Elementary Math Examples

3.6.8.1 Addition and Subtraction

▸ Show Section

Two-dimensional addition, subtraction, and multiplication typically involve numbers, carries/borrows, lines, and the sign of the operation.

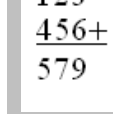
Below is the example shown at the start of the section: the digits inside the `mn` elements each occupy a column as does the

"+". `<mrow/>` is used to fill in the column under the "4" and make the "+" appear to the left of all of the operands. Notice that no attributes are given on `msline` causing it to span all of the columns.

```
<mstack>
  <mn>424</mn>
  <msrow> <mo>+</mo> <mrow/> <mn>33</mn> </msrow>
  <msline/>
</mstack>
```

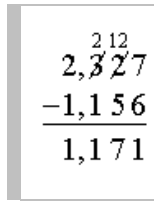
The next example illustrates how to put an operator on the right. Placing the operator on the right is standard in the Netherlands and some other countries. Notice that although there are a total of four columns in the example, because the default alignment is on the implied decimal point to the right of the numbers, it is not necessary to pad or shift any row.

```
<mstack>
  <mn>123</mn>
  <msrow> <mn>456</mn> <mo>+</mo> </msrow>
  <msline/>
  <mn>579</mn>
</mstack>
```



The following two examples illustrate the use of `mscarries`, `mscarry` and using `<mrow/>` to fill in a column. The examples also illustrate two different ways of displaying a borrow.

```
<mstack>
  <mscarries crossout='updiagonalstrike'>
    <mn>2</mn>  <mn>12</mn>  <mscarry crossout='none'> <mrow/> </mscarry>
  </mscarries>
  <mn>2,327</mn>
  <msrow> <mo>-</mo> <mn> 1,156</mn> </msrow>
  <msline/>
  <mn>1,171</mn>
</mstack>
```



```
<mstack>
  <mscarries location='nw'>
    <mrow/>
    <mscarry crossout='updiagonalstrike' location='n'> <mn>2</mn> </mscarry>
    <mn>1</mn>
    <mrow/>
  </mscarries>
  <mn>2,327</mn>
  <msrow> <mo>−</mo> <mn> 1,156</mn> </msrow>
  <msline/>
  <mn>1,171</mn>
</mstack>
```

$$\begin{array}{r} {}^{2}\not{3}^{1}27 \\ -1,156 \\ \hline 1,171 \end{array}$$

The MathML for the second example uses `mscarry` because a crossout should only happen on a single column:

The next example of subtraction shows a borrowed amount that is underlined (the example is from a Swedish source). There are two things to notice: an `menclose` is used in the carry, and `<mrow/>` is used for the empty element so that `mscarry` can be used to create a crossout.

```
<mstack>
  <mscarries>
    <mscarry crossout='updiagonalstrike'><mrow/></mscarry>
    <menclose notation='bottom'> <mn>10</mn> </menclose>
  </mscarries>
  <mn>52</mn>
  <msrow> <mo>−</mo> <mn> 7</mn> </msrow>
  <msline/>
  <mn>45</mn>
</mstack>
```

$$\begin{array}{r} {}^{10}\not{5}2 \\ -7 \\ \hline 45 \end{array}$$

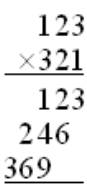
3.6.8.2 Multiplication

▸ Show Section

Below is a simple multiplication example that illustrates the use of `msgroup` and the `shift` attribute. The first `msgroup` is implied and doesn't change the layout. The second `msgroup` could also be removed, but `msrow` would be needed for last

two children. They `msrow` would need to set the `position` or `shift` attributes, or would add `<mrow/>` elements to pad the digits on the right.

```
<mstack>
  <msgroup>
    <mn>123</mn>
    <msrow><mo>×</mo><mn>321</mn></msrow>
  </msgroup>
  <msline/>
  <msgroup shift="1">
    <mn>123</mn>
    <mn>246</mn>
    <mn>369</mn>
  </msgroup>
  <msline/>
</mstack>
```

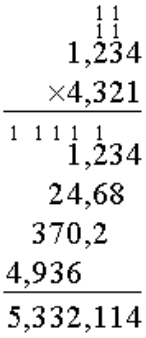


The following is a more complicated example of multiplication that has multiple rows of carries. It also (somewhat artificially) includes commas (",") as digit separators. The encoding includes these separators in the spacing attribute value, along non-ASCII values.

```
<mstack>
  <mscarries><mn>1</mn><mn>1</mn><mrow/></mscarries>
  <mscarries><mn>1</mn><mn>1</mn><mrow/></mscarries>
  <mn>1,234</mn>
  <msrow><mo>×</mo><mn>4,321</mn></msrow>
  <msline/>

  <mscarries position='2'>
    <mn>1</mn>
    <mrow/>
    <mn>1</mn>
    <mn>1</mn>
    <mn>1</mn>
    <mrow/>
    <mn>1</mn>
  </mscarries>
  <msgroup shift="1">
    <mn>1,234</mn>
    <mn>24,68</mn>
    <mn>370,2</mn>
    <msrow position="1"> <mn>4,936</mn> </msrow>
  </msgroup>
  <msline/>

  <mn>5,332,114</mn>
</mstack>
```



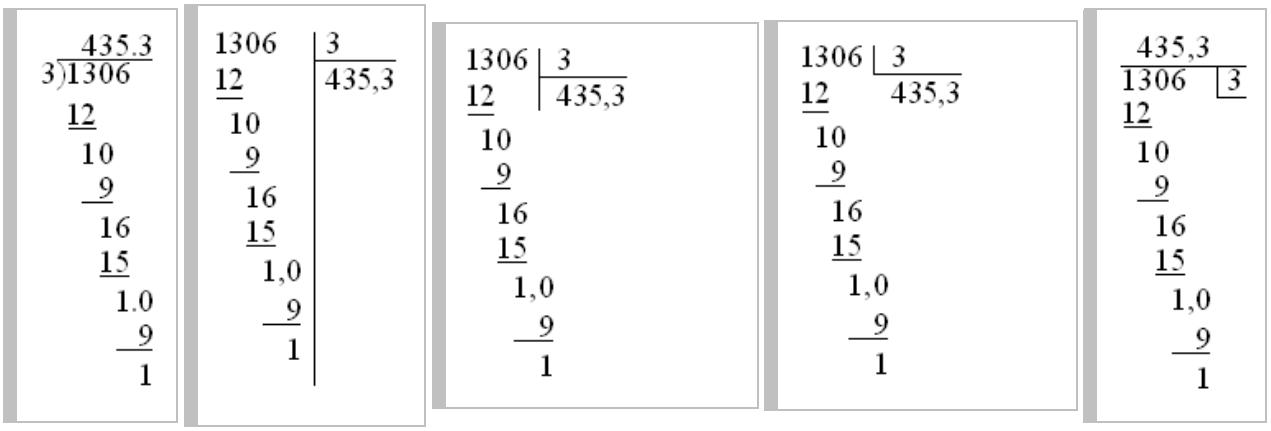
3.6.8.3 Long Division

▸ Show Section

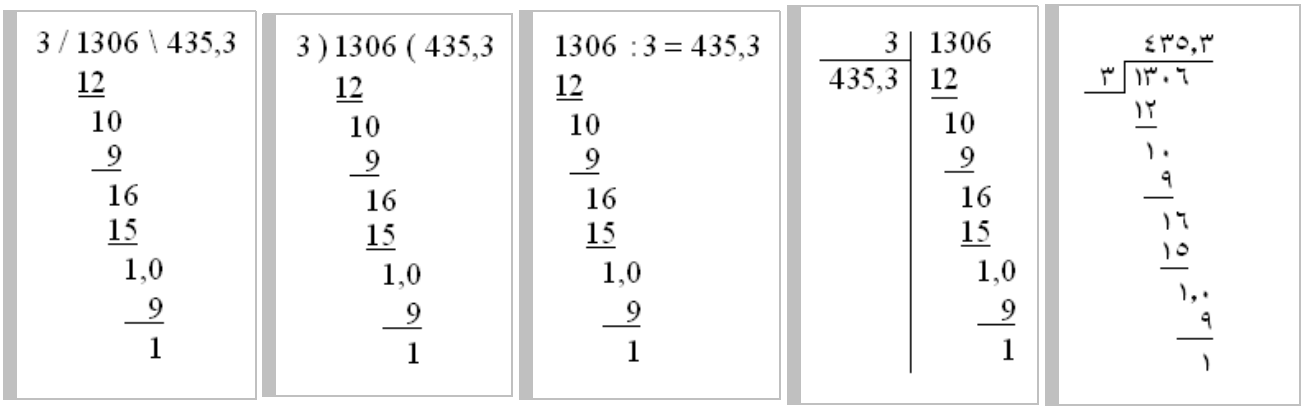
The notation used for long division varies considerably among countries. Most notations share the common characteristics of aligning intermediate results and drawing lines for the operands to be subtracted. Minus signs are sometimes shown for the intermediate calculations, and sometimes they are not. The line that is drawn varies in length depending upon the notation. The most apparent difference among the notations is that the position of the divisor varies, as does the location of the quotient, remainder, and intermediate terms.

The layout used is controlled by the `longdivstyle` attribute. Below are examples for the values listed in 3.6.2.2 Attributes.

`lefttop` `stackedrightright` `mediumstackedrightright` `shortstackedrightright` `righttop`



`left/\right` `left)(right` `:right=right` `stackedleftleft` `stackedleftlinetop`



The MathML for the first example is shown below. It illustrates the use of nested `msgroup`s and how the `position` is calculated in those usages.

```
<mlongdiv longdivstyle="lefttop">
  <mn> 3 </mn>
  <mn> 435.3</mn>

  <mn> 1306</mn>

  <msgroup position="2" shift="-1">
    <msgroup>
      <mn> 12</mn>
      <msline length="2"/>
    </msgroup>
    <msgroup>
      <mn> 10</mn>
      <mn> 9</mn>
      <msline length="2"/>
    </msgroup>
    <msgroup>
      <mn> 16</mn>
      <mn> 15</mn>
      <msline length="2"/>
      <mn> 1.0</mn>           <!-- aligns on '.', not the right edge ('0') -->
    </msgroup>
    <msgroup position='-1'>   <!-- extra shift to move to the right of the "." -->
      <mn> 9</mn>
      <msline length="3"/>
      <mn> 1</mn>
    </msgroup>
  </msgroup>
</mlongdiv>
```

```
  435.3
3)1306
  12
   10
   9
   16
   15
    1.0
    9
    1
```

With the exception of the last example, the encodings for the other examples are the same except that the values for `longdivstyle` differ and that a "," is used instead of a "." for the decimal point. For the last example, the only difference from the other examples besides a different value for `longdivstyle` is that Arabic numerals have been used in place of Latin numerals, as shown below.

```
<mstyle decimalpoint=",">
  <mlongdiv longdivstyle="stackedleftlinetop">
    <mn> ٣ </mn>
    <mn> ٤٣٥,٣</mn>

    <mn> ١٣٠٦</mn>
    <msgroup position="2" shift="-1">
      <msgroup>
        <mn> ١٢</mn>
        <msline length="2"/>
      </msgroup>
      <msgroup>
        <mn> ١٠</mn>
        <mn> ٩</mn>
        <msline length="2"/>
      </msgroup>
      <msgroup>
        <mn> ١٦</mn>
        <mn> ١٥</mn>
        <msline length="2"/>
        <mn> ١,٠</mn>
      </msgroup>
      <msgroup position='-1'>
        <mn> ٩</mn>
        <msline length="3"/>
        <mn> ١</mn>
      </msgroup>
    </msgroup>
  </mlongdiv>
</mstyle>
```

```
    ٤٣٥,٣
  ┌──────
٣ │ ١٣٠٦
    ١٢
    ──
     ١٠
      ٩
     ──
      ١٦
      ١٥
      ──
       ١,٠
         ٩
        ──
         ١
```

3.6.8.4 Repeating decimal

▸ Show Section

Decimal numbers that have digits that repeat infinitely such as 1/3 (.3333...) are represented using several notations. One common notation is to put a horizontal line over the digits that repeat (in Portugal an underline is used). Another notation

involves putting dots over the digits that repeat. The MathML for these involves using `mstack`, `msrow`, and `msline` in a straightforward manner. These notations are shown below:

```
<mstack stackalign="right">
  <msline length="1"/>
  <mn> 0.3333 </mn>
</mstack>
```

```
<mstack stackalign="right">
  <msline length="6"/>
  <mn> 0.142857 </mn>
</mstack>
```

```
<mstack stackalign="right">
  <mn> 0.142857 </mn>
  <msline length="6"/>
</mstack>
```

```
<mstack stackalign="right">
  <msrow> <mo>.</mo> <mrow/><mrow/><mrow/><mrow/> <mo>.</mo> </msrow>
  <mn> 0.142857 </mn>
</mstack>
```

3.7 Enlivening Expressions

3.7.1 Bind Action to Sub-Expression

The `maction` element provides a mechanism for binding actions to expressions. This element accepts any number of sub-expressions as arguments and the type of action that should happen is controlled by the `actiontype` attribute. MathML 3 predefined the four actions: `toggle`, `statusline`, `statusline`, and `input`. However, because the ability to implement any action depends very strongly on the platform, MathML 4 no longer predefines what these actions do. Furthermore, in the

web environment events connected to javascript to perform actions are a more powerful solution, although `maction` provides a convenient wrapper element on which to attach such an event.

Linking to other elements, either locally within the `math` element or to some URL, is not handled by `maction`. Instead, it is handled by adding a link directly on a MathML element as specified in 7.4.4 Linking.

3.7.1.1 Attributes

`maction` elements accept the attributes listed below in addition to those specified in 3.1.9 Mathematics attributes common to presentation elements.

By default, MathML applications that do not recognize the specified `actiontype` , or if the `actiontype` attribute is not present, should render the selected sub-expression as defined below. If no selected sub-expression exists, it is a MathML error; the appropriate rendering in that case is as described in D.2 Handling of Errors.

Name	values	default
actiontype[not-core]	*string*	
	Specifies what should happen for this element. The values allowed are open-ended. Conforming renderers may ignore any value they do not handle.	
selection[not-core]	*positive-integer*	1
	Specifies which child should be used for viewing. Its value should be between 1 and the number of children of the element. The specified child is referred to as the “selected sub-expression” of the `maction` element. If the value specified is out of range, it is an error. When the `selection` attribute is not specified (including for action types for which it makes no sense), its default value is 1, so the selected sub-expression will be the first sub-expression.	

If a MathML application responds to a user command to copy a MathML sub-expression to the environment's “clipboard” (see 7.3 Transferring MathML), any `maction` elements present in what is copied should be given `selection` values that correspond to their selection state in the MathML rendering at the time of the copy command.

When a MathML application receives a mouse event that may be processed by two or more nested maction elements, the innermost maction element of each action type should respond to the event.

The `actiontype` values are open-ended. If another value is given and it requires additional attributes, which should begin with "data-" or in XML they may be in a different namespace.

<maction actiontype="highlight" data-color="red" data-background="yellow"> expression </maction>
In the example, non-standard data attributes are being used to pass additional information to renderers that support them. The `data-color` attributes might change the color of the characters in the presentation, while the `data-background` attribute might change the color of the background behind the characters.

3.8 Semantics and Presentation

MathML uses the `semantics` element to allow specifying semantic annotations to presentation MathML elements; these can be content MathML or other notations. As such, `semantics` should be considered part of both presentation MathML and content MathML. All MathML processors should process the `semantics` element, even if they only process one of those subsets.

In semantic annotations a presentation MathML expression is typically the first child of the `semantics` element. However, it can also be given inside of an `annotation-xml` element inside the `semantics` element. If it is part of an `annotation-xml` element, then `encoding=application/mathml-presentation+xml` or `encoding=MathML-Presentation` may be used and presentation MathML processors should use this value for the presentation.

See 6. Annotating MathML: semantics for more details about the `semantics` and `annotation-xml` elements.

4. Content Markup

> ISSUE 284: Make the sample presentation of Strict Content use intent **MathML 4** **need specification update**
>
> There are currently "sample" renderings. Let's make this use `intent`.

4.1 Introduction

4.1.1 The Purpose of Content Markup

The purpose of Content Markup is to provide an explicit encoding of the *underlying mathematical meaning* of an expression, rather than any particular notation for the expression. Mathematical notation is at times ambiguous, context-dependent, and varies from community to community. In many cases, it is preferable to work directly with the underlying, formal, mathematical objects. Content Markup provides a rigorous, extensible semantic framework and a markup language for this purpose.

By encoding the underlying mathematical structure explicitly, without regard to how it is presented, it is possible to interchange information more precisely between systems that semantically process mathematical objects. Important application areas include computer algebra systems, automatic reasoning systems, industrial and scientific applications, multi-lingual translation systems, mathematical search, automated scoring of online assessments, and interactive textbooks.

This chapter presents an overview of basic concepts used to define Content Markup, describes a core collection of elements that comprise Strict Content Markup, and defines a full collection of elements to support common mathematical idioms. Strict Content Markup encodes general expression trees in a semantically rigorous way, while the full set of Content MathML elements provides backward-compatibility with previous versions of Content Markup. The correspondence between full Content Markup and Strict Content Markup is defined in F. The Strict Content MathML Transformation, which details an algorithm to translate arbitrary Content Markup into Strict Content Markup.

4.1.2 Content Expressions

Content MathML represents mathematical objects as *expression trees*. In general, an expression tree is constructed by applying an operator to a sequence of sub-expressions. For example, the sum "$x+y$" can be constructed as the application of the addition operator to two arguments x and y, and the expression "cos(π)" as the application of the cosine function to the number π.

The terminal nodes in an expression tree represent basic mathematical objects such as numbers, variables, arithmetic operations, and so on. The internal nodes in the tree represent function application or other mathematical constructions that build up compound objects.

MathML defines a relatively small number of commonplace mathematical constructs, chosen to be sufficient in a wide range of applications. In addition, it provides a mechanism to refer to concepts outside of the collection it defines, allowing them to be represented as well.

The defined set of content elements is designed to be adequate for simple coding of formulas typically used from kindergarten through the first two years of college in the United States, that is, up to A-Level or Baccalaureate level in Europe.

The primary role of the MathML content element set is to encode the mathematical structure of an expression independent of the notation used to present it. However, rendering issues cannot be ignored. There are many different approaches to render Content MathML formulae, ranging from native implementations of the MathML elements, to declarative notation definitions, to XSLT style sheets. Because rendering requirements for Content MathML vary widely, MathML does not provide a normative rendering specification. Instead, typical renderings are suggested by way of examples given using presentation markup.

4.1.3 Expression Concepts

The basic building blocks of Content MathML expressions are *numbers*, *identifiers*, and *symbols*. These building blocks are combined using function application and binding operators.

In the expression "$x + 2$", the numeral "2" represents a *number* with a fixed value. Content MathML uses the `cn` element to represent numerical quantities. The *identifier* "x" is a mathematical variable, that is, an identifier that represents a quantity with no predetermined value. Content MathML uses the `ci` element to represent variable identifiers.

The plus sign is an identifier that represents a fixed, externally defined object, namely, the addition function. Such an identifier is called a *symbol*, to distinguish it from a variable. Common elementary functions and operators are all symbols in this sense. Content MathML uses the `csymbol` element to represent symbols.

The fundamental way to combine numbers, variables, and symbols is function application. Content MathML distinguishes between the function itself (which may be a symbol such as the sine function, a variable such as f, or some other expression) and the result of applying the function to its arguments. The `apply` element groups the function with its arguments syntactically, and represents the expression that results from applying the function to its arguments.

4.1.4 Variable Binding

In an expression, variables may be described as *bound* or *free* variables. Bound variables have a special role within the scope of a binding expression, and may be renamed consistently within that scope without changing the meaning of the expression. Free variables are those that are not bound within an expression. Content MathML differentiates between the application of a function to a free variable (e.g. $f(x)$) and an operation that binds a variable within a binding scope. The `bind` element is used to delineate the binding scope of a bound variable and to group the binding operator with its bound variables, which are supplied using the `bvar` element.

In Strict Content markup, the only way to perform variable binding is to use the `bind` element. In non-Strict Content markup, other markup elements are provided that more closely resemble well-known idiomatic notations, such as "limit"-style notations for sums and integrals. These constructs may implicitly bind variables, such as the variable of integration, or the index variable in a sum. MathML uses the term *qualifier element* to refer to those elements used to represent the auxiliary data required by these constructs.

Expressions involving qualifiers follow one of a small number of idiomatic patterns, each of which applies to a class of similar binding operators. For example, sums and products are in the same class because they use index variables following the same pattern. The Content MathML operator classes are described in detail in 4.3.4 Operator Classes.

4.1.5 Strict Content MathML

Beginning in MathML 3, *Strict Content MathML* is defined as a minimal subset of Content MathML that is sufficient to represent the meaning of mathematical expressions using a uniform structure. The full Content MathML element set retains backward compatibility with MathML 2, and strikes a pragmatic balance between verbosity and formality.

Content MathML provides a considerable number of predefined functions encoded as empty elements (e.g. `sin`, `log`, etc.) and a variety of constructs for forming compound objects (e.g. `set`, `interval`, etc.). In contrast, Strict Content MathML represents all known functions using a single element (`csymbol`) with an attribute that points to its definition in an extensible content dictionary, and uses only `apply` and `bind` elements to build up compound expressions. Token elements such as `cn` and `ci` are considered part of Strict Content MathML, but with a more restricted set of attributes and with content restricted to text.

The formal semantics of Content MathML expressions are given by specifying equivalent Strict Content MathML expressions, which all have formal semantics defined in terms of content dictionaries. The exact correspondence between each non-Strict Content MathML structure and its Strict Content MathML equivalent is described in terms of rewrite rules that are used as part of the transformation algorithm given in F. The Strict Content MathML Transformation.

The algorithm described in F. The Strict Content MathML Transformation is complete in the sense that it gives every Content MathML expression a specific meaning in terms of a Strict Content MathML expression. In some cases, it gives a specific strict interpretation to an expression whose meaning was not sufficiently specified in MathML 2. The goal of this algorithm is to be faithful to natural mathematical intuitions, however, some edge cases may remain where the specific interpretation given by the algorithm may be inconsistent with earlier expectations.

A conformant MathML processor need not implement this algorithm. The existence of these transformation rules does not imply that a system must treat equivalent expressions identically. In particular, systems may give different presentation

renderings for expressions that the transformation rules imply are mathematically equivalent. In general, Content MathML does not define any expectations for the computational behavior of the expressions it encodes, including, but not limited to, the equivalence of any specific expressions.

Strict Content MathML is designed to be compatible with OpenMath, a standard for representing formal mathematical objects and semantics. Strict Content MathML is an XML encoding of OpenMath Objects in the sense of [OpenMath]. The following table gives the correspondence between Strict Content MathML elements and their OpenMath equivalents.

Strict Content MathML	OpenMath
cn	OMI, OMF
csymbol	OMS
ci	OMV
cs	OMSTR
apply	OMA
bind	OMBIND
bvar	OMBVAR
share	OMR
semantics	OMATTR
annotation, annotation-xml	OMATP, OMFOREIGN
cerror	OME
cbytes	OMB

4.1.6 Content Dictionaries

Any method to formalize the meaning of mathematical expressions must be extensible, that is, it must provide the ability to define new functions and symbols to expand the domain of discourse. Content MathML uses the csymbol element to represent new symbols, and uses *Content Dictionaries* to describe their mathematical semantics. The association between a symbol and its semantic description is accomplished using the attributes of the csymbol element to point to the definition of the symbol in a Content Dictionary.

The correspondence between operator elements in Content MathML and symbol definitions in Content Dictionaries is given in E.3 The Content MathML Operators. These definitions for predefined MathML operator symbols refer to Content Dictionaries developed by the OpenMath Society [OpenMath] in conjunction with the W3C Math Working Group. It is important to note that this information is informative, not normative. In general, the precise mathematical semantics of predefined symbols are not fully specified by the MathML Recommendation, and the only normative statements about symbol semantics are those present in the text of this chapter. The semantic definitions provided by the OpenMath Content Dictionaries are intended to be sufficient for most applications, and are generally compatible with the semantics specified for analogous constructs in this Recommendation. However, in contexts where highly precise semantics are required (e.g. communication between computer algebra systems, within formal systems such as theorem provers, etc.) it is the responsibility of the relevant community of practice to verify, extend or replace definitions provided by OpenMath Content Dictionaries as appropriate.

4.2 Content MathML Elements Encoding Expression Structure

In this section we will present the elements for encoding the structure of content MathML expressions. These elements are the only ones used for the Strict Content MathML encoding. Concretely, we have

- basic expressions, i.e. Numbers, string literals, encoded bytes, Symbols, and Identifiers.
- derived expressions, i.e. function applications and binding expressions, and
- semantic annotations
- error markup

Full Content MathML allows further elements presented in 4.3 Content MathML for Specific Structures and 4.3 Content MathML for Specific Structures, and allows a richer content model presented in this section. Differences in Strict and non-Strict usage of are highlighted in the sections discussing each of the Strict element below.

4.2.1 Numbers <*cn*>

	Schema Fragment (Strict)		Schema Fragment (Full)	
Class	Cn		Cn	
Attributes	`CommonAtt`, `type`		`CommonAtt`, `DefEncAtt`, `type`?, `base`?	
`type` Attribute Values	`integer` | `real` | `double` | `hexdouble`		`integer` | `real` | `double` | `hexdouble` | `e-notation` | `rational` | `complex-cartesian` | `complex-polar` | `constant` | text	default is real
`base` Attribute Values			`integer`	default is 10
Content	text		(text | `mglyph` | `sep` | `PresentationExpression`)*	

The `cn` element is the Content MathML element used to represent numbers. Strict Content MathML supports integers, real numbers, and double precision floating point numbers. In these types of numbers, the content of `cn` is text. Additionally, `cn` supports rational numbers and complex numbers in which the different parts are separated by use of the `sep` element. Constructs using `sep` may be rewritten in Strict Content MathML as constructs using `apply` as described below.

The `type` attribute specifies which kind of number is represented in the `cn` element. The default value is `real`. Each type implies that the content be of a certain form, as detailed below.

4.2.1.1 Rendering <cn>,<sep/>-Represented Numbers

The default rendering of the text content of `cn` is the same as that of the Presentation element `mn`, with suggested variants in

the case of attributes or `sep` being used, as listed below.

4.2.1.2 Strict uses of `<cn>`

In Strict Content MathML, the `type` attribute is mandatory, and may only take the values `integer`, `real`, `hexdouble` or `double`:

integer
An integer is represented by an optional sign followed by a string of one or more decimal "digits".

real
A real number is presented in radix notation. Radix notation consists of an optional sign ("+" or "-") followed by a string of digits possibly separated into an integer and a fractional part by a decimal point. Some examples are 0.3, 1, and -31.56.

double
This type is used to mark up those double-precision floating point numbers that can be represented in the IEEE 754 standard format [IEEE754]. This includes a subset of the (mathematical) real numbers, negative zero, positive and negative real infinity and a set of "not a number" values. The lexical rules for interpreting the text content of a `cn` as an IEEE double are specified by Section 3.1.2.5 of XML Schema Part 2: Datatypes Second Edition [XMLSchemaDatatypes]. For example, -1E4, 1267.43233E12, 12.78e-2, 12, -0, 0 and INF are all valid doubles in this format.

hexdouble
This type is used to directly represent the 64 bits of an IEEE 754 double-precision floating point number as a 16 digit hexadecimal number. Thus the number represents mantissa, exponent, and sign from lowest to highest bits using a least significant byte ordering. This consists of a string of 16 digits 0-9, A-F. The following example represents a NaN value. Note that certain IEEE doubles, such as the NaN in the example, cannot be represented in the lexical format for the `double` type.

```
<cn type="hexdouble">7F800000</cn>
```

Sample Presentation

```
<mn>0x7F800000</mn>
```

0x7F800000

4.2.1.3 Non-Strict uses of `<cn>`

The `base` attribute is used to specify how the content is to be parsed. The attribute value is a base 10 positive integer giving the value of base in which the text content of the `cn` is to be interpreted. The `base` attribute should only be used on elements

with type `integer` or `real`. Its use on `cn` elements of other type is deprecated. The default value for `base` is `10`.

Additional values for the `type` attribute element for supporting e-notations for real numbers, rational numbers, complex numbers and selected important constants. As with the `integer`, `real`, `double` and `hexdouble` types, each of these types implies that the content be of a certain form. If the `type` attribute is omitted, it defaults to `real`.

integer

Integers can be represented with respect to a base different from 10: If `base` is present, it specifies (in base 10) the base for the digit encoding. Thus `base`='16' specifies a hexadecimal encoding. When `base` > 10, Latin letters (A-Z, a-z) are used in alphabetical order as digits. The case of letters used as digits is not significant. The following example encodes the base 10 number 32736.

```
<cn base="16">7FE0</cn>
```

Sample Presentation

```
<msub><mn>7FE0</mn><mn>16</mn></msub>
```

$7FE0_{16}$

When `base` > 36, some integers cannot be represented using numbers and letters alone. For example, while

```
<cn base="1000">10F</cn>
```

arguably represents the number written in base 10 as 1,000,015, the number written in base 10 as 1,000,037 cannot be represented using letters and numbers alone when `base` is 1000. Consequently, support for additional characters (if any) that may be used for digits when `base` > 36 is application specific.

real

Real numbers can be represented with respect to a base different than 10. If a `base` attribute is present, then the digits are interpreted as being digits computed relative to that base (in the same way as described for type `integer`).

e-notation

A real number may be presented in scientific notation using this type. Such numbers have two parts (a significand and an exponent) separated by a `<sep/>` element. The first part is a real number, while the second part is an integer exponent indicating a power of the base.

For example, `<cn type="e-notation">12.3<sep/>5</cn>` represents 12.3×10^5. The default presentation of this example is 12.3e5. Note that this type is primarily useful for backwards compatibility with MathML 2, and in most cases, it is preferable to use the `double` type, if the number to be represented is in the range of IEEE doubles:

rational

A rational number is given as two integers to be used as the numerator and denominator of a quotient. The numerator and denominator are separated by `<sep/>`.

```
<cn type="rational">22<sep/>7</cn>
```

Sample Presentation

```
<mrow><mn>22</mn><mo>/</mo><mn>7</mn></mrow>
```

$$22 / 7$$

complex-cartesian

A complex cartesian number is given as two numbers specifying the real and imaginary parts. The real and imaginary parts are separated by the <sep/> element, and each part has the format of a real number as described above.

```
<cn type="complex-cartesian"> 12.3 <sep/> 5 </cn>
```

Sample Presentation

```
<mrow>
  <mn>12.3</mn><mo>+</mo><mn>5</mn><mo>&#x2062;<!--InvisibleTimes--></mo><mi>i</mi>
</mrow>
```

$$12.3 + 5i$$

complex-polar

A complex polar number is given as two numbers specifying the magnitude and angle. The magnitude and angle are separated by the <sep/> element, and each part has the format of a real number as described above.

```
<cn type="complex-polar"> 2 <sep/> 3.1415 </cn>
```

Sample Presentation

```
<mrow>
  <mn>2</mn>
  <mo>&#x2062;<!--InvisibleTimes--></mo>
  <msup>
    <mi>e</mi>
    <mrow><mi>i</mi><mo>&#x2062;<!--InvisibleTimes--></mo><mn>3.1415</mn></mrow>
  </msup>
</mrow>
```

$$2e^{i3.1415}$$

```
<mrow>
  <mi>Polar</mi>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow><mo>(</mo><mn>2</mn><mo>,</mo><mn>3.1415</mn><mo>)</mo></mrow>
</mrow>
```

$$\mathrm{Polar}\,(2, 3.1415)$$

constant

If the value `type` is `constant`, then the content should be a Unicode representation of a well-known constant. Some important constants and their common Unicode representations are listed below.

This `cn` type is primarily for backward compatibility with MathML 1.0. MathML 2.0 introduced many empty elements, such as `<pi/>` to represent constants, and using these representations or a Strict csymbol representation is preferred.

In addition to the additional values of the type attribute, the content of `cn` element can contain (in addition to the `sep` element allowed in Strict Content MathML) `mglyph` elements to refer to characters not currently available in Unicode, or a general presentation construct (see 3.1.8 Summary of Presentation Elements), which is used for rendering (see 4.1.2 Content Expressions).

If a `base` attribute is present, it specifies the base used for the digit encoding of both integers. The use of `base` with `rational` numbers is deprecated.

4.2.2 Content Identifiers <*ci*>

	Schema Fragment (Strict)	Schema Fragment (Full)
Class	Ci	Ci
Attributes	`CommonAtt`, `type`?	`CommonAtt`, `DefEncAtt`, `type`?
`type` Attribute Values	`integer`| `rational`| `real`| `complex`| `complex-polar`| `complex-cartesian`| `constant`| `function`| `vector`| `list`| `set`| `matrix`	string
Qualifiers		`BvarQ`, `DomainQ`, `degree`, `momentabout`, `logbase`
Content	text	text | `mglyph` | `PresentationExpression`

Content MathML uses the `ci` element (mnemonic for "content identifier") to construct a variable. Content identifiers represent "mathematical variables" which have properties, but no fixed value. For example, x and y are variables in the expression "$x+y$", and the variable x would be represented as

```
<ci>x</ci>
```

In MathML, variables are distinguished from symbols, which have fixed, external definitions, and are represented by the

csymbol element.

After white space normalization the content of a `ci` element is interpreted as a name that identifies it. Two variables are considered equal, if and only if their names are identical and in the same scope (see 4.2.6 Bindings and Bound Variables <bind> and <bvar> for a discussion).

4.2.2.1 Strict uses of `<ci>`

The `ci` element uses the `type` attribute to specify the basic type of object that it represents. In Strict Content MathML, the set of permissible values is `integer`, `rational`, `real`, `complex`, `complex-polar`, `complex-cartesian`, `constant`, `function`, `vector`, `list`, `set`, and `matrix`. These values correspond to the symbols integer_type, rational_type, real_type, complex_polar_type, complex_cartesian_type, constant_type, fn_type, vector_type, list_type, set_type, and matrix_type in the mathmltypes Content Dictionary: In this sense the following two expressions are considered equivalent:

```
<ci type="integer">n</ci>
```

```
<semantics>
  <ci>n</ci>
  <annotation-xml cd="mathmltypes" name="type" encoding="MathML-Content">
    <csymbol cd="mathmltypes">integer_type</csymbol>
  </annotation-xml>
</semantics>
```

Note that `complex` should be considered an alias for `complex-cartesian` and rewritten to the same complex_cartesian_type symbol. It is perhaps a more natural type name for use with `ci` as the distinction between cartesian and polar form really only affects the interpretation of literals encoded with `cn`.

4.2.2.2 Non-Strict uses of `<ci>`

The `ci` element allows any string value for the `type` attribute, in particular any of the names of the MathML container elements or their type values.

For a more advanced treatment of types, the `type` attribute is inappropriate. Advanced types require significant structure of their own (for example, *vector(complex)*) and are probably best constructed as mathematical objects and then associated with a MathML expression through use of the `semantics` element. See [MathML-Types] for more examples.

4.2.2.3 Rendering Content Identifiers

If the content of a `ci` element consists of Presentation MathML, that presentation is used. If no such tagging is supplied then

the text content is rendered as if it were the content of an `mi` element. If an application supports bidirectional text rendering, then the rendering follows the Unicode bidirectional rendering.

The `type` attribute can be interpreted to provide rendering information. For example in

```
<ci type="vector">V</ci>
```

a renderer could display a bold V for the vector.

4.2.3 Content Symbols <csymbol>

	Schema Fragment (Strict)	Schema Fragment (Full)
Class	Csymbol	Csymbol
Attributes	CommonAtt, cd	CommonAtt, DefEncAtt, type?, cd?
Content	SymbolName	text | mglyph | PresentationExpression
Qualifiers		BvarQ, DomainQ, degree, momentabout, logbase

A `csymbol` is used to refer to a specific, mathematically-defined concept with an external definition. In the expression "$x+y$", the plus sign is a symbol since it has a specific, external definition, namely the addition function. MathML 3 calls such an identifier a *symbol*. Elementary functions and common mathematical operators are all examples of symbols. Note that the term "symbol" is used here in an abstract sense and has no connection with any particular presentation of the construct on screen or paper.

4.2.3.1 Strict uses of <csymbol>

The `csymbol` identifies the specific mathematical concept it represents by referencing its definition via attributes. Conceptually, a reference to an external definition is merely a URI, i.e. a label uniquely identifying the definition. However, to be useful for communication between user agents, external definitions must be shared.

For this reason, several longstanding efforts have been organized to develop systematic, public repositories of mathematical definitions. Most notable of these, the OpenMath Society repository of Content Dictionaries (CDs) is extensive, open and active. In MathML 3, OpenMath CDs are the preferred source of external definitions. In particular, the definitions of pre-defined MathML 3 operators and functions are given in terms of OpenMath CDs.

MathML 3 provides two mechanisms for referencing external definitions or content dictionaries. The first, using the `cd` attribute, follows conventions established by OpenMath specifically for referencing CDs. This is the form required in Strict Content MathML. The second, using the `definitionURL` attribute, is backward compatible with MathML 2, and can be used to reference CDs or any other source of definitions that can be identified by a URI. It is described in the following section.

When referencing OpenMath CDs, the preferred method is to use the `cd` attribute as follows. Abstractly, OpenMath symbol definitions are identified by a triple of values: a *symbol name*, a *CD name*, and a *CD base*, which is a URI that disambiguates

CDs of the same name. To associate such a triple with a `csymbol`, the content of the `csymbol` specifies the symbol name, and the name of the Content Dictionary is given using the `cd` attribute. The CD base is determined either from the document embedding the `math` element which contains the `csymbol` by a mechanism given by the embedding document format, or by system defaults, or by the `cdgroup` attribute, which is optionally specified on the enclosing `math` element; see 2.2.1 Attributes. In the absence of specific information `http://www.openmath.org/cd` is assumed as the CD base for all `csymbol` elements `annotation`, and `annotation-xml`. This is the CD base for the collection of standard CDs maintained by the OpenMath Society.

The `cdgroup` specifies a URL to an OpenMath CD Group file. For a detailed description of the format of a CD Group file, see Section 4.4.2 (CDGroups) in [OpenMath]. Conceptually, a CD group file is a list of pairs consisting of a CD name, and a corresponding CD base. When a `csymbol` references a CD name using the `cd` attribute, the name is looked up in the CD Group file, and the associated CD base value is used for that `csymbol`. When a CD Group file is specified, but a referenced CD name does not appear in the group file, or there is an error in retrieving the group file, the referencing `csymbol` is not defined. However, the handling of the resulting error is not defined, and is the responsibility of the user agent.

While references to external definitions are URIs, it is strongly recommended that CD files be retrievable at the location obtained by interpreting the URI as a URL. In particular, other properties of the symbol being defined may be available by inspecting the Content Dictionary specified. These include not only the symbol definition, but also examples and other formal properties. Note, however, that there are multiple encodings for OpenMath Content Dictionaries, and it is up to the user agent to correctly determine the encoding when retrieving a CD.

4.2.3.2 Non-Strict uses of `<csymbol>`

In addition to the forms described above, the `csymbol` and element can contain `mglyph` elements to refer to characters not currently available in Unicode, or a general presentation construct (see 3.1.8 Summary of Presentation Elements), which is used for rendering (see 4.1.2 Content Expressions). In this case, when writing to Strict Content MathML, the csymbol should be treated as a `ci` element, and rewritten using Rewrite: ci presentation mathml.

External definitions (in OpenMath CDs or elsewhere) may also be specified directly for a `csymbol` using the `definitionURL` attribute. When used to reference OpenMath symbol definitions, the abstract triple of (symbol name, CD name, CD base) is mapped to a fully-qualified URI as follows:

```
URI = cdbase + '/' + cd-name + '#' + symbol-name
```

For example,

```
(plus, arith1, http://www.openmath.org/cd)
```

is mapped to

```
http://www.openmath.org/cd/arith1#plus
```

The resulting URI is specified as the value of the `definitionURL` attribute.

This form of reference is useful for backwards compatibility with MathML2 and to facilitate the use of Content MathML

within URI-based frameworks (such as RDF [RDF] in the Semantic Web or OMDoc [OMDoc1.2]). Another benefit is that the symbol name in the CD does not need to correspond to the content of the `csymbol` element. However, in general, this method results in much longer MathML instances. Also, in situations where CDs are under development, the use of a CD Group file allows the locations of CDs to change without a change to the markup. A third drawback to `definitionURL` is that unlike the `cd` attribute, it is not limited to referencing symbol definitions in OpenMath content dictionaries. Hence, it is not in general possible for a user agent to automatically determine the proper interpretation for `definitionURL` values without further information about the context and community of practice in which the MathML instance occurs.

Both the `cd` and `definitionURL` mechanisms of external reference may be used within a single MathML instance. However, when both a `cd` and a `definitionURL` attribute are specified on a single `csymbol`, the `cd` attribute takes precedence.

4.2.3.3 Rendering Symbols

If the content of a `csymbol` element is tagged using presentation tags, that presentation is used. If no such tagging is supplied then the text content is rendered as if it were the content of an `mi` element. In particular if an application supports bidirectional text rendering, then the rendering follows the Unicode bidirectional rendering.

4.2.4 String Literals <cs>

	Schema Fragment (Strict)	Schema Fragment (Full)
Class	Cs	Cs
Attributes	`CommonAtt`	`CommonAtt`, `DefEncAtt`
Content	text	text

The `cs` element encodes “string literals” which may be used in Content MathML expressions.

The content of cs is text; no Presentation MathML constructs are allowed even when used in non-strict markup. Specifically, `cs` may not contain `mglyph` elements, and the content does not undergo white space normalization.

Content MathML

```
<set>
  <cs>A</cs><cs>B</cs><cs>  </cs>
</set>
```

Sample Presentation

```
<mrow>
  <mo>{</mo>
  <ms>A</ms>
  <mo>,</mo>
  <ms>B</ms>
  <mo>,</mo>
  <ms>  </ms>
  <mo>}</mo>
</mrow>
```

{"A","B"," "}

4.2.5 Function Application <*apply*>

	Schema Fragment (Strict)	Schema Fragment (Full)
Class	Apply	Apply
Attributes	CommonAtt	CommonAtt, DefEncAtt
Content	ContExp+	ContExp+ | (ContExp, BvarQ, Qualifier?, ContExp*)

The most fundamental way of building a compound object in mathematics is by applying a function or an operator to some arguments.

4.2.5.1 Strict Content MathML

In MathML, the `apply` element is used to build an expression tree that represents the application of a function or operator to its arguments. The resulting tree corresponds to a complete mathematical expression. Roughly speaking, this means a piece of mathematics that could be surrounded by parentheses or “logical brackets” without changing its meaning.

For example, $(x + y)$ might be encoded as

```
<apply><csymbol cd="arith1">plus</csymbol><ci>x</ci><ci>y</ci></apply>
```

The opening and closing tags of `apply` specify exactly the scope of any operator or function. The most typical way of using `apply` is simple and recursive. Symbolically, the content model can be described as:

```
<apply> op [ a b ...] </apply>
```

where the *operands* *a*, *b*, ... are MathML expression trees themselves, and *op* is a MathML expression tree that represents an operator or function. Note that `apply` constructs can be nested to arbitrary depth.

An `apply` may in principle have any number of operands. For example, $(x + y + z)$ can be encoded as

```
<apply><csymbol cd="arith1">plus</csymbol>
  <ci>x</ci>
  <ci>y</ci>
  <ci>z</ci>
</apply>
```

Note that MathML also allows applications without operands, e.g. to represent functions like `random()`, or `current-date()`.

Mathematical expressions involving a mixture of operations result in nested occurrences of `apply`. For example, $a\ x + b$ would be encoded as

```
<apply><csymbol cd="arith1">plus</csymbol>
  <apply><csymbol cd="arith1">times</csymbol>
    <ci>a</ci>
    <ci>x</ci>
  </apply>
  <ci>b</ci>
</apply>
```

There is no need to introduce parentheses or to resort to operator precedence in order to parse expressions correctly. The `apply` tags provide the proper grouping for the re-use of the expressions within other constructs. Any expression enclosed by an `apply` element is well-defined, coherent object whose interpretation does not depend on the surrounding context. This is in sharp contrast to presentation markup, where the same expression may have very different meanings in different contexts. For example, an expression with a visual rendering such as $(F+G)(x)$ might be a product, as in

```
<apply><csymbol cd="arith1">times</csymbol>
  <apply><csymbol cd="arith1">plus</csymbol>
    <ci>F</ci>
    <ci>G</ci>
  </apply>
  <ci>x</ci>
</apply>
```

or it might indicate the application of the function $F + G$ to the argument x. This is indicated by constructing the sum

```
<apply><csymbol cd="arith1">plus</csymbol><ci>F</ci><ci>G</ci></apply>
```

and applying it to the argument x as in

```
<apply>
  <apply><csymbol cd="arith1">plus</csymbol>
    <ci>F</ci>
    <ci>G</ci>
  </apply>
  <ci>x</ci>
</apply>
```

In both cases, the interpretation of the outer `apply` is explicit and unambiguous, and does not change regardless of where the expression is used.

The preceding example also illustrates that in an `apply` construct, both the function and the arguments may be simple identifiers or more complicated expressions.

The `apply` element is conceptually necessary in order to distinguish between a function or operator, and an instance of its use. The expression constructed by applying a function to 0 or more arguments is always an element from the codomain of the function. Proper usage depends on the operator that is being applied. For example, the `plus` operator may have zero or more arguments, while the `minus` operator requires one or two arguments in order to be properly formed.

§ *4.2.5.2 Rendering Applications*

Strict Content MathML applications are rendered as mathematical function applications. If `<mi>F</mi>` denotes the rendering of `<ci>f</ci>` and `<mi>Ai</mi>` the rendering of `<ci>ai</ci>`, the sample rendering of a simple application is as follows:

Content MathML

```
<apply><ci>f</ci>
  <ci>a1</ci>
  <ci>a2</ci>
  <ci>...</ci>
  <ci>an</ci>
</apply>
```

Sample Presentation

```
<mrow>
  <mi>F</mi>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
    <mo fence="true">(</mo>
    <mi>A1</mi>
    <mo separator="true">,</mo>
    <mi>...</mi>
    <mo separator="true">,</mo>
    <mi>A2</mi>
    <mo separator="true">,</mo>
    <mi>An</mi>
    <mo fence="true">)</mo>
  </mrow>
</mrow>
```

$$F(\mathrm{A1}, ..., \mathrm{A2}, \mathrm{An})$$

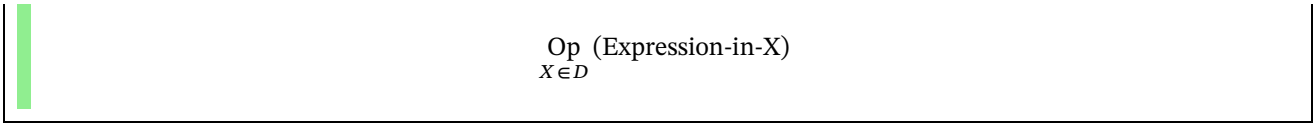
Non-Strict MathML applications may also be used with qualifiers. In the absence of any more specific rendering rules for well-known operators, rendering should follow the sample presentation below, motivated by the typical presentation for `sum`. Let `<mi>Op</mi>` denote the rendering of `<ci>op</ci>`, `<mi>X</mi>` the rendering of `<ci>x</ci>`, and so on. Then:

Content MathML

```
<apply><ci>op</ci>
  <bvar><ci>x</ci></bvar>
  <domainofapplication><ci>d</ci></domainofapplication>
  <ci>expression-in-x</ci>
</apply>
```

Sample Presentation

```
<mrow>
  <munder>
    <mi>Op</mi>
    <mrow><mi>X</mi><mo>∈</mo><mi>D</mi></mrow>
  </munder>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
    <mo fence="true">(</mo>
    <mi>Expression-in-X</mi>
    <mo fence="true">)</mo>
  </mrow>
</mrow>
```



4.2.6 Bindings and Bound Variables <*bind*> and <*bvar*>

Many complex mathematical expressions are constructed with the use of bound variables, and bound variables are an important concept of logic and formal languages. Variables become *bound* in the scope of an expression through the use of a quantifier. Informally, they can be thought of as the "dummy variables" in expressions such as integrals, sums, products, and the logical quantifiers "for all" and "there exists". A bound variable is characterized by the property that systematically renaming the variable (to a name not already appearing in the expression) does not change the meaning of the expression.

4.2.6.1 Bindings

	Schema Fragment (Strict)	Schema Fragment (Full)
Class	Bind	Bind
Attributes	`CommonAtt`	`CommonAtt`, `DefEncAtt`
Content	`ContExp`, `BvarQ*`, `ContExp`	`ContExp`, `BvarQ*`, `Qualifier*`, `ContExp+`

Binding expressions are represented as MathML expression trees using the `bind` element. Its first child is a MathML expression that represents a binding operator, for example integral operator. This is followed by a non-empty list of `bvar` elements denoting the bound variables, and then the final child which is a general Content MathML expression, known as the *body* of the binding.

4.2.6.2 Bound Variables

	Schema Fragment (Strict)	Schema Fragment (Full)
Class	BVar	BVar
Attributes	`CommonAtt`	`CommonAtt`, `DefEncAtt`
Content	`ci	semantics-ci`	`(ci | semantics-ci)`, `degree?	degree?`, `(ci | semantics-ci)`

The `bvar` element is used to denote the bound variable of a binding expression, e.g. in sums, products, and quantifiers or user defined functions.

The content of a `bvar` element is an *annotated variable*, i.e. either a content identifier represented by a `ci` element or a `semantics` element whose first child is an annotated variable. The *name* of an annotated variable of the second kind is the name of its first child. The *name* of a bound variable is that of the annotated variable in the `bvar` element.

Bound variables are identified by comparing their names. Such identification can be made explicit by placing an `id` on the

`ci` element in the `bvar` element and referring to it using the `xref` attribute on all other instances. An example of this approach is

```
<bind><csymbol cd="quant1">forall</csymbol>
  <bvar><ci id="var-x">x</ci></bvar>
  <apply><csymbol cd="relation1">lt</csymbol>
    <ci xref="var-x">x</ci>
    <cn>1</cn>
  </apply>
</bind>
```

This `id` based approach is especially helpful when constructions involving bound variables are nested.

It is sometimes necessary to associate additional information with a bound variable. The information might be something like a detailed mathematical type, an alternative presentation or encoding or a domain of application. Such associations are accomplished in the standard way by replacing a `ci` element (even inside the `bvar` element) by a `semantics` element containing both the `ci` and the additional information. Recognition of an instance of the bound variable is still based on the actual `ci` elements and not the `semantics` elements or anything else they may contain. The `id`-based approach outlined above may still be used.

The following example encodes $\forall x\,.\,x + y = y + x$.

```
<bind><csymbol cd="quant1">forall</csymbol>
  <bvar><ci>x</ci></bvar>
  <apply><csymbol cd="relation1">eq</csymbol>
    <apply><csymbol cd="arith1">plus</csymbol><ci>x</ci><ci>y</ci></apply>
    <apply><csymbol cd="arith1">plus</csymbol><ci>y</ci><ci>x</ci></apply>
  </apply>
</bind>
```

In non-Strict Content markup, the `bvar` element is used in a number of idiomatic constructs. These are described in 4.3.3 Qualifiers and 4.3 Content MathML for Specific Structures.

4.2.6.3 Renaming Bound Variables

It is a defining property of bound variables that they can be renamed consistently in the scope of their parent `bind` element. This operation, sometimes known as *α-conversion*, preserves the semantics of the expression.

A bound variable x may be renamed to say y so long as y does not occur free in the body of the binding, or in any annotations of the bound variable, x to be renamed, or later bound variables.

If a bound variable x is renamed, all free occurrences of x in annotations in its `bvar` element, any following `bvar` children of the `bind` and in the expression in the body of the `bind` should be renamed.

In the example in the previous section, note how renaming x to z produces the equivalent expression $\forall z\,.\,z + y = y + z$, whereas x may not be renamed to y, as y is free in the body of the binding and would be *captured*, producing the expression $\forall y\,.\,y + y = y + y$ which is not equivalent to the original expression.

4.2.6.4 Rendering Binding Constructions

If `<ci>b</ci>` and `<ci>s</ci>` are Content MathML expressions that render as the Presentation MathML expressions `<mi>B</mi>` and `<mi>S</mi>` then the sample rendering of a binding element is as follows:

Content MathML

```
<bind><ci>b</ci>
  <bvar><ci>x1</ci></bvar>
  <bvar><ci>...</ci></bvar>
  <bvar><ci>xn</ci></bvar>
  <ci>s</ci>
</bind>
```

Sample Presentation

```
<mrow>
  <mi>B</mi>
  <mrow>
    <mi>x1</mi>
    <mo separator="true">,</mo>
    <mi>...</mi>
    <mo separator="true">,</mo>
    <mi>xn</mi>
  </mrow>
  <mo separator="true">.</mo>
  <mi>S</mi>
</mrow>
```

$$B x1, ..., xn . S$$

4.2.7 Structure Sharing `<share>`

To conserve space in the XML encoding, MathML expression trees can make use of structure sharing.

4.2.7.1 The `share` element

	Schema Fragment
Class	Share
Attributes	`CommonAtt`, `src`
`src` Attribute Values	*URI*
Content	Empty

The `share` element has an `src` attribute used to reference a MathML expression tree. The value of the `src` attribute is a URI specifying the `id` attribute of the root node of the expression tree. When building a MathML expression tree, the `share` element is equivalent to a copy of the MathML expression tree referenced by the `src` attribute. Note that this copy is *structurally equal*, but not identical to the element referenced. The values of the `share` will often be relative URI references, in which case they are resolved using the base URI of the document containing the `share` element.

For instance, the mathematical object *f(f(f(a,a),f(a,a)),f(f(a,a),f(a,a)))* can be encoded as either one of the following representations (and some intermediate versions as well).

```
<apply><ci>f</ci>
  <apply><ci>f</ci>
    <apply><ci>f</ci>
      <ci>a</ci>
      <ci>a</ci>
    </apply>
    <apply><ci>f</ci>
      <ci>a</ci>
      <ci>a</ci>
    </apply>
  </apply>
  <apply><ci>f</ci>
    <apply><ci>f</ci>
      <ci>a</ci>
      <ci>a</ci>
    </apply>
    <apply><ci>f</ci>
      <ci>a</ci>
      <ci>a</ci>
    </apply>
  </apply>
</apply>
```

```
<apply><ci>f</ci>
  <apply id="t1"><ci>f</ci>
    <apply id="t11"><ci>f</ci>
      <ci>a</ci>
      <ci>a</ci>
    </apply>
    <share src="#t11"/>


  </apply>
  <share src="#t1"/>









</apply>
```

4.2.7.2 An Acyclicity Constraint

Say that an element *dominates* all its children and all elements they dominate. Say also that a `share` element dominates its target, i.e. the element that carries the `id` attribute pointed to by the `src` attribute. For instance in the representation on the right above, the `apply` element with `id="t1"` and also the second `share` (with `src="t11"`) both dominate the `apply`

element with `id="t11"`.

The occurrences of the `share` element must obey the following global *acyclicity constraint*: An element may not dominate itself. For example, the following representation violates this constraint:

```
<apply id="badid1"><csymbol cd="arith1">divide</csymbol>
  <cn>1</cn>
  <apply><csymbol cd="arith1">plus</csymbol>
    <cn>1</cn>
    <share src="#badid1"/>
  </apply>
</apply>
```

Here, the `apply` element with `id="badid1"` dominates its third child, which dominates the `share` element, which dominates its target: the element with `id="badid1"`. So by transitivity, this element dominates itself. By the acyclicity constraint, the example is not a valid MathML expression tree. It might be argued that such an expression could be given the interpretation of the continued fraction 1 / (1 + 1 / (1 + ⋯ . However, the procedure of building an expression tree by replacing `share` element does not terminate for such an expression, and hence such expressions are not allowed by Content MathML.

Note that the acyclicity constraint is not restricted to such simple cases, as the following example shows:

```
<apply id="bar">                        <apply id="baz">
  <csymbol cd="arith1">plus</csymbol>     <csymbol cd="arith1">plus</csymbol>
  <cn>1</cn>                              <cn>1</cn>
  <share src="#baz"/>                    <share src="#bar"/>
</apply>                                </apply>
```

Here, the `apply` with `id="bar"` dominates its third child, the `share` with `src="#baz"`. That element dominates its target `apply` (with `id="baz"`), which in turn dominates its third child, the `share` with `src="#bar"`. Finally, the `share` with `src="#bar"` dominates its target, the original `apply` element with `id="bar"`. So this pair of representations ultimately violates the acyclicity constraint.

4.2.7.3 Structure Sharing and Binding

Note that the `share` element is a *syntactic* referencing mechanism: a `share` element stands for the exact element it points to. In particular, referencing does not interact with binding in a semantically intuitive way, since it allows a phenomenon called *variable capture* to occur. Consider an example:

```
<bind id="outer"><csymbol cd="fns1">lambda</csymbol>
  <bvar><ci>x</ci></bvar>
  <apply><ci>f</ci>
    <bind id="inner"><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <share id="copy" src="#orig"/>
    </bind>
    <apply id="orig"><ci>g</ci><ci>x</ci></apply>
  </apply>
</bind>
```

This represents a term $\lambda x\,.\,f(\lambda x\,.\,g(x), g(x))$ which has two sub-terms of the form $g(x)$, one with `id="orig"` (the one explicitly represented) and one with `id="copy"`, represented by the `share` element. In the original, explicitly-represented term, the variable x is bound by the *outer* `bind` element. However, in the copy, the variable x is bound by the *inner* `bind` element. One says that the inner `bind` has captured the variable x.

Using references that capture variables in this way can easily lead to representation errors, and is not recommended. For instance, using α-conversion to rename the inner occurrence of x into, say, y leads to the semantically equivalent expression $\lambda x\,.\,f(\lambda y\,.\,g(y), g(x))$. However, in this form, it is no longer possible to share the expression $g(x)$. Replacing x with y in the inner `bvar` without replacing the `share` element results in a change in semantics.

4.2.7.4 Rendering Expressions with Structure Sharing

There are several acceptable renderings for the `share` element. These include rendering the element as a hypertext link to the referenced element and using the rendering of the element referenced by the `src` attribute.

4.2.8 Attribution via `semantics`

Content elements can be annotated with additional information via the `semantics` element. MathML uses the `semantics` element to wrap the annotated element and the `annotation-xml` and `annotation` elements used for representing the annotations themselves. The use of the `semantics`, `annotation` and `annotation-xml` is described in detail in 6. Annotating MathML: semantics.

The `semantics` element is considered part of both presentation MathML and Content MathML. MathML considers a `semantics` element (strict) Content MathML, if and only if its first child is (strict) Content MathML.

4.2.9 Error Markup `<cerror>`

	Schema Fragment (Strict)	Schema Fragment (Full)
Class	Error	Error
Attributes	`CommonAtt`	`CommonAtt`, `DefEncAtt`
Content	`csymbol`, `ContExp*`	`csymbol`, `ContExp*`

A content error expression is made up of a `csymbol` followed by a sequence of zero or more MathML expressions. The initial expression must be a `csymbol` indicating the kind of error. Subsequent children, if present, indicate the context in which the error occurred.

The `cerror` element has no direct mathematical meaning. Errors occur as the result of some action performed on an expression tree and are thus of real interest only when some sort of communication is taking place. Errors may occur inside other objects and also inside other errors.

As an example, to encode a division by zero error, one might employ a hypothetical `aritherror` Content Dictionary containing a `DivisionByZero` symbol, as in the following expression:

```
<cerror>
  <csymbol cd="aritherror">DivisionByZero</csymbol>
  <apply><csymbol cd="arith1">divide</csymbol><ci>x</ci><cn>0</cn></apply>
</cerror>
```

Note that error markup generally should enclose only the smallest erroneous sub-expression. Thus a `cerror` will often be a sub-expression of a bigger one, e.g.

```
<apply><csymbol cd="relation1">eq</csymbol>
  <cerror>
    <csymbol cd="aritherror">DivisionByZero</csymbol>
    <apply><csymbol cd="arith1">divide</csymbol><ci>x</ci><cn>0</cn></apply>
  </cerror>
  <cn>0</cn>
</apply>
```

The default presentation of a `cerror` element is an `merror` expression whose first child is a presentation of the error symbol, and whose subsequent children are the default presentations of the remaining children of the `cerror`. In particular, if one of the remaining children of the `cerror` is a presentation MathML expression, it is used literally in the corresponding `merror`.

```
<cerror>
  <csymbol cd="aritherror">DivisionByZero</csymbol>
  <apply><csymbol cd="arith1">divide</csymbol><ci>x</ci><cn>0</cn></apply>
</cerror>
```

Sample Presentation

```
<merror>
  <mtext>DivisionByZero: </mtext>
  <mfrac><mi>x</mi><mn>0</mn></mfrac>
</merror>
```

DivisionByZero: $\frac{x}{0}$

Note that when the context where an error occurs is so nonsensical that its default presentation would not be useful, an application may provide an alternative representation of the error context. For example:

```
<cerror>
  <csymbol cd="error">Illegal bound variable</csymbol>
  <cs> &lt;bvar&gt;&lt;plus/&gt;&lt;/bvar&gt; </cs>
</cerror>
```

4.2.10 Encoded Bytes <*cbytes*>

	Schema Fragment (Strict)	Schema Fragment (Full)
Class	Cbytes	Cbytes
Attributes	`CommonAtt`	`CommonAtt, DefEncAtt`
Content	`base64`	`base64`

The content of `cbytes` represents a stream of bytes as a sequence of characters in Base64 encoding, that is it matches the base64Binary data type defined in [XMLSchemaDatatypes]. All white space is ignored.

The `cbytes` element is mainly used for OpenMath compatibility, but may be used, as in OpenMath, to encapsulate output from a system that may be hard to encode in MathML, such as binary data relating to the internal state of a system, or image data.

The rendering of `cbytes` is not expected to represent the content and the proposed rendering is that of an empty `mrow`. Typically `cbytes` is used in an `annotation-xml` or is itself annotated with Presentation MathML, so this default rendering should rarely be used.

4.3 Content MathML for Specific Structures

The elements of Strict Content MathML described in the previous section are sufficient to encode logical assertions and expression structure, and they do so in a way that closely models the standard constructions of mathematical logic that underlie the foundations of mathematics. As a consequence, Strict markup can be used to represent all of mathematics, and is ideal for providing consistent mathematical semantics for all Content MathML expressions.

At the same time, many notational idioms of mathematics are not straightforward to represent directly with Strict Content markup. For example, standard notations for sums, integrals, sets, piecewise functions and many other common constructions require non-obvious technical devices, such as the introduction of lambda functions, to rigorously encode them using Strict markup. Consequently, in order to make Content MathML easier to use, a range of additional elements have been provided for encoding such idiomatic constructs more directly. This section discusses the general approach for encoding such idiomatic constructs, and their Strict Content equivalents. Specific constructions are discussed in detail in 4.3 Content MathML for Specific Structures.

Most idiomatic constructions which Content markup addresses fall into about a dozen classes. Some of these classes, such as *container elements*, have their own syntax. Similarly, a small number of non-Strict constructions involve a single element with an exceptional syntax, for example `partialdiff`. These exceptional elements are discussed on a case-by-case basis in 4.3 Content MathML for Specific Structures. However, the majority of constructs consist of classes of operator elements which all share a particular usage of *qualifiers*. These classes of operators are described in 4.3.4 Operator Classes.

In all cases, non-Strict expressions may be rewritten using only Strict markup. In most cases, the transformation is completely algorithmic, and may be automated. Rewrite rules for classes of non-Strict constructions are introduced and discussed later in this section, and rewrite rules for exceptional constructs involving a single operator are given in 4.3 Content MathML for Specific Structures. The complete algorithm for rewriting arbitrary Content MathML as Strict Content markup is summarized at the end of the Chapter in F. The Strict Content MathML Transformation.

4.3.1 Container Markup

Many mathematical structures are constructed from subparts or parameters. For example, a set is a mathematical object that contains a collection of elements, so it is natural for the markup for a set to contain the markup for its constituent elements. The markup for a set may define the set of elements explicitly by enumerating them, or implicitly by rule that uses qualifier elements. In either case, the markup for the elements is contained in the markup for the set, and this style of representation is called *container markup* in MathML. By contrast, Strict markup represents an instance of a set as the result of applying a function or *constructor symbol* to arguments. In this style of markup, the markup for the set construction is a sibling of the markup for the set elements in an enclosing `apply` element.

MathML provides container markup for the following mathematical constructs: sets, lists, intervals, vectors, matrices (two elements), piecewise functions (three elements) and lambda functions. There are corresponding constructor symbols in Strict markup for each of these, with the exception of lambda functions, which correspond to binding symbols in Strict markup.

The rewrite rules for obtaining equivalent Strict Content markup from container markup depend on the operator class of the particular operator involved. For details about a specific container element, obtain its operator class (and any applicable special case information) by consulting the syntax table and discussion for that element in E. The Content MathML Operators. Then apply the rewrite rules for that specific operator class as described in F. The Strict Content MathML Transformation.

4.3.1.1 Container Markup for Constructor Symbols

The arguments to container elements that correspond to constructors may be explicitly given as a sequence of child elements, or implicitly given by a rule using qualifiers. The exceptions are the `interval`, `piecewise`, `piece`, and `otherwise` elements. The arguments of these elements must be specified explicitly.

Here is an example of container markup with explicitly specified arguments:

```
<set><ci>a</ci><ci>b</ci><ci>c</ci></set>
```

This is equivalent to the following Strict Content MathML expression:

```
<apply><csymbol cd="set1">set</csymbol><ci>a</ci><ci>b</ci><ci>c</ci></apply>
```

Another example of container markup, where the list of arguments is given indirectly as an expression with a bound variable. The container markup for the set of even integers is:

```
<set>
  <bvar><ci>x</ci></bvar>
  <domainofapplication><integers/></domainofapplication>
  <apply><times/><cn>2</cn><ci>x</ci></apply>
</set>
```

This may be written as follows in Strict Content MathML:

```
<apply><csymbol cd="set1">map</csymbol>
  <bind><csymbol cd="fns1">lambda</csymbol>
    <bvar><ci>x</ci></bvar>
    <apply><csymbol cd="arith1">times</csymbol>
      <cn>2</cn>
      <ci>x</ci>
    </apply>
  </bind>
  <csymbol cd="setname1">Z</csymbol>
</apply>
```

4.3.1.2 Container Markup for Binding Constructors

The `lambda` element is a container element corresponding to the lambda symbol in the fns1 Content Dictionary. However, unlike the container elements of the preceding section, which purely construct mathematical objects from arguments, the `lambda` element performs variable binding as well. Therefore, the child elements of `lambda` have distinguished roles. In particular, a `lambda` element must have at least one `bvar` child, optionally followed by qualifier elements, followed by a Content MathML element. This basic difference between the `lambda` container and the other constructor container elements is also reflected in the OpenMath symbols to which they correspond. The constructor symbols have an OpenMath role of "application", while the lambda symbol has a role of "bind".

This example shows the use of `lambda` container element and the equivalent use of `bind` in Strict Content MathML

```
<lambda><bvar><ci>x</ci></bvar><ci>x</ci></lambda>
```

```
<bind><csymbol cd="fns1">lambda</csymbol>
  <bvar><ci>x</ci></bvar><ci>x</ci>
</bind>
```

4.3.2 Bindings with <apply>

MathML allows the use of the `apply` element to perform variable binding in non-Strict constructions instead of the `bind` element. This usage conserves backwards compatibility with MathML 2. It also simplifies the encoding of several constructs involving bound variables with qualifiers as described below.

Use of the `apply` element to bind variables is allowed in two situations. First, when the operator to be applied is itself a binding operator, the `apply` element merely substitutes for the `bind` element. The logical quantifiers `<forall/>`, `<exists/>` and the container element `lambda` are the primary examples of this type.

The second situation arises when the operator being applied allows the use of bound variables with qualifiers. The most common examples are sums and integrals. In most of these cases, the variable binding is to some extent implicit in the notation, and the equivalent Strict representation requires the introduction of auxiliary constructs such as lambda expressions for formal correctness.

Because expressions using bound variables with qualifiers are idiomatic in nature, and do not always involve true variable binding, one cannot expect systematic renaming (alpha-conversion) of variables “bound” with `apply` to preserve meaning in all cases. An example for this is the `diff` element where the `bvar` term is technically not bound at all.

The following example illustrates the use of `apply` with a binding operator. In these cases, the corresponding Strict equivalent merely replaces the `apply` element with a `bind` element:

```
<apply><forall/>
  <bvar><ci>x</ci></bvar>
  <apply><geq/><ci>x</ci><ci>x</ci></apply>
</apply>
```

The equivalent Strict expression is:

```
<bind><csymbol cd="logic1">forall</csymbol>
  <bvar><ci>x</ci></bvar>
  <apply><csymbol cd="relation1">geq</csymbol><ci>x</ci><ci>x</ci></apply>
</bind>
```

In this example, the sum operator is not itself a binding operator, but bound variables with qualifiers are implicit in the standard notation, which is reflected in the non-Strict markup. In the equivalent Strict representation, it is necessary to

convert the summand into a lambda expression, and recast the qualifiers as an argument expression:

```
<apply><sum/>
  <bvar><ci>i</ci></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <uplimit><cn>100</cn></uplimit>
  <apply><power/><ci>x</ci><ci>i</ci></apply>
</apply>
```

The equivalent Strict expression is:

```
<apply><csymbol cd="arith1">sum</csymbol>
  <apply><csymbol cd="interval1">integer_interval</csymbol>
    <cn>0</cn>
    <cn>100</cn>
  </apply>
  <bind><csymbol cd="fns1">lambda</csymbol>
    <bvar><ci>i</ci></bvar>
    <apply><csymbol cd="arith1">power</csymbol>
      <ci>x</ci>
      <ci>i</ci>
    </apply>
  </bind>
</apply>
```

4.3.3 Qualifiers

Many common mathematical constructs involve an operator together with some additional data. The additional data is either implicit in conventional notation, such as a bound variable, or thought of as part of the operator, as is the case with the limits of a definite integral. MathML 3 uses *qualifier* elements to represent the additional data in such cases.

Qualifier elements are always used in conjunction with operator or container elements. Their meaning is idiomatic, and depends on the context in which they are used. When used with an operator, qualifiers always follow the operator and precede any arguments that are present. In all cases, if more than one qualifier is present, they appear in the order `bvar`, `lowlimit`, `uplimit`, `interval`, `condition`, `domainofapplication`, `degree`, `momentabout`, `logbase`.

The precise function of qualifier elements depends on the operator or container that they modify. The majority of use cases fall into one of several categories, discussed below, and usage notes for specific operators and qualifiers are given in 4.3 Content MathML for Specific Structures.

4.3.3.1 Uses of **<domainofapplication>**, **<interval>**, **<condition>**, **<lowlimit>** *and* **<uplimit>**

Class	qualifier
Attributes	CommonAtt
Content	ContExp

(For the syntax of `interval` see 4.3.10.3 Interval <interval>.)

The primary use of `domainofapplication`, `interval`, `uplimit`, `lowlimit` and `condition` is to restrict the values of a bound variable. The most general qualifier is `domainofapplication`. It is used to specify a set (perhaps with additional structure, such as an ordering or metric) over which an operation is to take place. The `interval` qualifier, and the pair `lowlimit` and `uplimit` also restrict a bound variable to a set in the special case where the set is an interval. Note that `interval` is only interpreted as a qualifier if it immediately follows `bvar`. The `condition` qualifier, like `domainofapplication`, is general, and can be used to restrict bound variables to arbitrary sets. However, unlike the other qualifiers, it restricts the bound variable by specifying a Boolean-valued function of the bound variable. Thus, `condition` qualifiers always contain instances of the bound variable, and thus require a preceding `bvar`, while the other qualifiers do not. The other qualifiers may even be used when no variables are being bound, e.g. to indicate the restriction of a function to a subdomain.

In most cases, any of the qualifiers capable of representing the domain of interest can be used interchangeably. The most general qualifier is `domainofapplication`, and therefore has a privileged role. It is the preferred form, unless there are particular idiomatic reasons to use one of the other qualifiers, e.g. limits for an integral. In MathML 3, the other forms are treated as shorthand notations for `domainofapplication` because they may all be rewritten as equivalent `domainofapplication` constructions. The rewrite rules to do this are given below. The other qualifier elements are provided because they correspond to common notations and map more easily to familiar presentations. Therefore, in the situations where they naturally arise, they may be more convenient and direct than `domainofapplication`.

To illustrate these ideas, consider the following examples showing alternative representations of a definite integral. Let C denote the interval from 0 to 1, and $f(x) = x^2$. Then `domainofapplication` could be used to express the integral of a function f over C in this way:

```
<apply><int/>
  <domainofapplication>
    <ci type="set">C</ci>
  </domainofapplication>
  <ci type="function">f</ci>
</apply>
```

Note that no explicit bound variable is identified in this encoding, and the integrand is a function. Alternatively, the `interval` qualifier could be used with an explicit bound variable:

```
<apply><int/>
  <bvar><ci>x</ci></bvar>
  <interval><cn>0</cn><cn>1</cn></interval>
  <apply><power/><ci>x</ci><cn>2</cn></apply>
</apply>
```

The pair `lowlimit` and `uplimit` can also be used. This is perhaps the most “standard” representation of this integral:

```
<apply><int/>
  <bvar><ci>x</ci></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <uplimit><cn>1</cn></uplimit>
  <apply><power/><ci>x</ci><cn>2</cn></apply>
</apply>
```

Finally, here is the same integral, represented using a `condition` on the bound variable:

```
<apply><int/>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><and/>
      <apply><leq/><cn>0</cn><ci>x</ci></apply>
      <apply><leq/><ci>x</ci><cn>1</cn></apply>
    </apply>
  </condition>
  <apply><power/><ci>x</ci><cn>2</cn></apply>
</apply>
```

Note the use of the explicit bound variable within the `condition` term. Note also that when a bound variable is used, the integrand is an expression in the bound variable, not a function.

The general technique of using a `condition` element together with `domainofapplication` is quite powerful. For example, to extend the previous example to a multivariate domain, one may use an extra bound variable and a domain of application corresponding to a cartesian product:

```
<apply><int/>
  <bvar><ci>x</ci></bvar>
  <bvar><ci>y</ci></bvar>
  <domainofapplication>
    <set>
      <bvar><ci>t</ci></bvar>
      <bvar><ci>u</ci></bvar>
      <condition>
        <apply><and/>
          <apply><leq/><cn>0</cn><ci>t</ci></apply>
          <apply><leq/><ci>t</ci><cn>1</cn></apply>
          <apply><leq/><cn>0</cn><ci>u</ci></apply>
          <apply><leq/><ci>u</ci><cn>1</cn></apply>
        </apply>
      </condition>
      <list><ci>t</ci><ci>u</ci></list>
    </set>
  </domainofapplication>
  <apply><times/>
    <apply><power/><ci>x</ci><cn>2</cn></apply>
    <apply><power/><ci>y</ci><cn>3</cn></apply>
  </apply>
</apply>
```

Note that the order of the inner and outer bound variables is significant.

§ *4.3.3.2 Uses of* ***<degree>***

Class	qualifier
Attributes	CommonAtt
Content	ContExp

The degree element is a qualifier used to specify the “degree” or “order” of an operation. MathML uses the degree element in this way in three contexts: to specify the degree of a root, a moment, and in various derivatives. Rather than introduce special elements for each of these families, MathML provides a single general construct, the degree element in all three cases.

Note that the degree qualifier is not used to restrict a bound variable in the same sense of the qualifiers discussed above. Indeed, with roots and moments, no bound variable is involved at all, either explicitly or implicitly. In the case of differentiation, the degree element is used in conjunction with a bvar, but even in these cases, the variable may not be genuinely bound.

For the usage of degree with the root and moment operators, see the discussion of those operators below. The usage of degree in differentiation is more complex. In general, the degree element indicates the order of the derivative with respect to that variable. The degree element is allowed as the second child of a bvar element identifying a variable with respect to which the derivative is being taken. Here is an example of a second derivative using the degree qualifier:

```
<apply><diff/>
  <bvar>
    <ci>x</ci>
    <degree><cn>2</cn></degree>
  </bvar>
  <apply><power/><ci>x</ci><cn>4</cn></apply>
</apply>
```

For details see 4.3.8.2 Differentiation <diff/> and 4.3.8.3 Partial Differentiation <partialdiff/>.

4.3.3.3 Uses of <momentabout> *and* <logbase>

The qualifiers momentabout and logbase are specialized elements specifically for use with the moment and log operators respectively. See the descriptions of those operators below for their usage.

4.3.4 Operator Classes

The Content MathML elements described in detail in the following sections may be broadly separated into *classes*. The class of each element is listed in the operator syntax table given in E.3 The Content MathML Operators. The class gives an indication of the general intended mathematical usage of the element, and also determines its usage as determined by the schema. Links to the operator syntax and schema class for each element are provided in the sections that introduce the elements.

The operator class also determines the applicable rewrite rules for mapping to Strict Content MathML. These rewrite rules are presented in detail in F. The Strict Content MathML Transformation. They include use cases applicable to specific operator classes, special-case rewrite rules for individual elements, and a generic rewrite rule F.8 Rewrite operators used by operators from almost all operator classes.

The following sections present elements representing a core set of mathematical operators, functions and constants. Most are empty elements, covering the subject matter of standard mathematics curricula up to the level of calculus. The remaining elements are container elements for sets, intervals, vectors and so on. For brevity, all elements defined in this section are sometimes called *operator elements*.

4.3.5 N-ary Operators

Many MathML operators may be used with an arbitrary number of arguments. The corresponding OpenMath symbols for elements in these classes also take an arbitrary number of arguments. In all such cases, either the arguments may be given explicitly as children of the apply or bind element, or the list may be specified implicitly via the use of qualifier elements.

4.3.5.1 N-ary Arithmetic Operators: **<plus/>**, **<times/>**, **<gcd/>**, **<lcm/>**

Operator Syntax, Schema Class

The `plus` and `times` elements represent the addition and multiplication operators. The arguments are normally specified explicitly in the enclosing `apply` element. As an n-ary commutative operator, they can be used with qualifiers to specify arguments, however, this is discouraged, and the `sum` or `product` operators should be used to represent such expressions instead.

4.3.5.1.1 EXAMPLE

▸ Show Section

Content MathML

```
<apply><plus/><ci>x</ci><ci>y</ci><ci>z</ci></apply>
```

Sample Presentation

```
<mrow><mi>x</mi><mo>+</mo><mi>y</mi><mo>+</mo><mi>z</mi></mrow>
```

$$x + y + z$$

The `gcd` and `lcm` elements represent the n-ary operators which return the greatest common divisor, or least common multiple of their arguments. The arguments may be explicitly specified in the enclosing apply element, or specified by quantifiers.

This default renderings are English-language locale specific: other locales may have different default renderings.

4.3.5.1.2 EXAMPLE

▸ Show Section

Content MathML

```
<apply><gcd/><ci>a</ci><ci>b</ci><ci>c</ci></apply>
```

Sample Presentation

```
<mrow>
  <mi>gcd</mi>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow><mo>(</mo><mi>a</mi><mo>,</mo><mi>b</mi><mo>,</mo><mi>c</mi><mo>)</mo></mrow>
</mrow>
```

$$\gcd\left(a, b, c\right)$$

4.3.5.2 N-ary Sum **<sum/>**

Operator Syntax, Schema Class

The `sum` element represents the n-ary addition operator. The terms of the sum are normally specified by rule through the use of qualifiers. While it can be used with an explicit list of arguments, this is strongly discouraged, and the `plus` operator should be used instead in such situations.

The `sum` operator may be used either with or without explicit bound variables. When a bound variable is used, the `sum` element is followed by one or more `bvar` elements giving the index variables, followed by qualifiers giving the domain for the index variables. The final child in the enclosing `apply` is then an expression in the bound variables, and the terms of the sum are obtained by evaluating this expression at each point of the domain of the index variables. Depending on the structure of the domain, the domain of summation is often given by using `uplimit` and `lowlimit` to specify upper and lower limits for the sum.

When no bound variables are explicitly given, the final child of the enclosing `apply` element must be a function, and the terms of the sum are obtained by evaluating the function at each point of the domain specified by qualifiers.

4.3.5.2.1 Examples

▸ Show Section

Content MathML

```
<apply><sum/>
  <bvar><ci>x</ci></bvar>
  <lowlimit><ci>a</ci></lowlimit>
  <uplimit><ci>b</ci></uplimit>
  <apply><ci>f</ci><ci>x</ci></apply>
</apply>
```

```
<apply><sum/>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><in/><ci>x</ci><ci type="set">B</ci></apply>
  </condition>
  <apply><ci type="function">f</ci><ci>x</ci></apply>
</apply>
```

```
<apply><sum/>
  <domainofapplication>
    <ci type="set">B</ci>
  </domainofapplication>
  <ci type="function">f</ci>
</apply>
```

Sample Presentation

```
<mrow>
  <munderover>
    <mo>∑</mo>
    <mrow><mi>x</mi><mo>=</mo><mi>a</mi></mrow>
    <mi>b</mi>
  </munderover>
  <mrow><mi>f</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>x</mi><mo>)</n
</mrow>
```

$$\sum_{x=a}^{b} f(x)$$

```
<mrow>
  <munder>
    <mo>∑</mo>
    <mrow><mi>x</mi><mo>∈</mo><mi>B</mi></mrow>
  </munder>
  <mrow><mi>f</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>x</mi><mo>)</n
</mrow>
```

$$\sum_{x \in B} f(x)$$

```
<mrow><munder><mo>∑</mo><mi>B</mi></munder><mi>f</mi></mrow>
```

$$\sum_{B} f$$

4.3.5.3 N-ary Product **<product/>**

Operator Syntax, Schema Class

The `product` element represents the n-ary multiplication operator. The terms of the product are normally specified by rule through the use of qualifiers. While it can be used with an explicit list of arguments, this is strongly discouraged, and the `times` operator should be used instead in such situations.

The `product` operator may be used either with or without explicit bound variables. When a bound variable is used, the `product` element is followed by one or more `bvar` elements giving the index variables, followed by qualifiers giving the domain for the index variables. The final child in the enclosing `apply` is then an expression in the bound variables, and the terms of the product are obtained by evaluating this expression at each point of the domain. Depending on the structure of the domain, it is commonly given using `uplimit` and `lowlimit` qualifiers.

When no bound variables are explicitly given, the final child of the enclosing `apply` element must be a function, and the terms of the product are obtained by evaluating the function at each point of the domain specified by qualifiers.

4.3.5.3.1 Examples

▸ Show Section

Content MathML

```
<apply><product/>
  <bvar><ci>x</ci></bvar>
  <lowlimit><ci>a</ci></lowlimit>
  <uplimit><ci>b</ci></uplimit>
  <apply><ci type="function">f</ci>
    <ci>x</ci>
  </apply>
</apply>
```

```
<apply><product/>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><in/>
      <ci>x</ci>
      <ci type="set">B</ci>
    </apply>
  </condition>
  <apply><ci>f</ci><ci>x</ci></apply>
</apply>
```

Sample Presentation

```
<mrow>
  <munderover>
    <mo>∏</mo>
    <mrow><mi>x</mi><mo>=</mo><mi>a</mi></mrow>
    <mi>b</mi>
  </munderover>
  <mrow><mi>f</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>x</mi><mo>)</n
</mrow>
```

$$\prod_{x=a}^{b} f(x)$$

```
<mrow>
  <munder>
    <mo>∏</mo>
    <mrow><mi>x</mi><mo>∈</mo><mi>B</mi></mrow>
  </munder>
  <mrow><mi>f</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>x</mi><mo>)</n
</mrow>
```

$$\prod_{x \in B} f(x)$$

4.3.5.4 N-ary Functional Operators: **<compose/>**

Operator Syntax, Schema Class

The `compose` element represents the function composition operator. Note that MathML makes no assumption about the domain and codomain of the constituent functions in a composition; the domain of the resulting composition may be empty.

The `compose` element is a commutative n-ary operator. Consequently, it may be lifted to the induced operator defined on a collection of arguments indexed by a (possibly infinite) set by using qualifier elements as described in 4.3.5.4 N-ary Functional Operators: <compose/>.

4.3.5.4.1 Examples

▶ Show Section

Content MathML

```
<apply><compose/><ci>f</ci><ci>g</ci><ci>h</ci></apply>
```

```
<apply><eq/>
  <apply>
    <apply><compose/><ci>f</ci><ci>g</ci></apply>
    <ci>x</ci>
  </apply>
  <apply><ci>f</ci><apply><ci>g</ci><ci>x</ci></apply></apply>
</apply>
```

Sample Presentation

```
<mrow>
  <mi>f</mi><mo>∘</mo><mi>g</mi><mo>∘</mo><mi>h</mi>
</mrow>
```

$$f \circ g \circ h$$

```
<mrow>
  <mrow>
    <mrow><mo>(</mo><mi>f</mi><mo>∘</mo><mi>g</mi><mo>)</mo></mrow>
    <mo>&#x2061;<!--ApplyFunction--></mo>
    <mrow><mo>(</mo><mi>x</mi><mo>)</mo></mrow>
  </mrow>
  <mo>=</mo>
  <mrow>
    <mi>f</mi>
    <mo>&#x2061;<!--ApplyFunction--></mo>
    <mrow>
     <mo>(</mo>
      <mrow>
        <mi>g</mi>
        <mo>&#x2061;<!--ApplyFunction--></mo>
        <mrow><mo>(</mo><mi>x</mi><mo>)</mo></mrow>
      </mrow>
      <mo>)</mo>
    </mrow>
  </mrow>
</mrow>
```

$$(f \circ g)(x) = f(g(x))$$

§ *4.3.5.5 N-ary Logical Operators:* **<and/>**, **<or/>**, **<xor/>**

Operator Syntax, Schema Class

These elements represent n-ary functions taking Boolean arguments and returning a Boolean value. The arguments may be explicitly specified in the enclosing `apply` element, or specified via qualifier elements.

`and` is true if all arguments are true, and false otherwise.
`or` is true if any of the arguments are true, and false otherwise.
`xor` is the logical “exclusive or” function. It is true if there are an odd number of true arguments or false otherwise.

4.3.5.5.1 Examples

▸ Show Section

Content MathML

```
<apply><and/><ci>a</ci><ci>b</ci></apply>
```

```
<apply><and/>
  <bvar><ci>i</ci></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <uplimit><ci>n</ci></uplimit>
  <apply><gt/><apply><selector/><ci>a</ci><ci>i</ci></apply><cn>0</cn></apply>
</apply>
```

Strict Content MathML

```
<apply><csymbol cd="logic1">and</csymbol><ci>a</ci><ci>b</ci></apply>
```

```
<apply><csymbol cd="fns2">apply_to_list</csymbol>
  <csymbol cd="logic1">and</csymbol>
  <apply><csymbol cd="list1">map</csymbol>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>i</ci></bvar>
      <apply><csymbol cd="relation1">gt</csymbol>
        <apply><csymbol cd="linalg1">vector_selector</csymbol>
          <ci>i</ci>
          <ci>a</ci>
        </apply>
        <cn>0</cn>
      </apply>
    </bind>
    <apply><csymbol cd="interval1">integer_interval</csymbol>
      <cn type="integer">0</cn>
      <ci>n</ci>
    </apply>
  </apply>
</apply>
```

Sample Presentation

```
<mrow><mi>a</mi><mo>∧</mo><mi>b</mi></mrow>
```

$$a \wedge b$$

```
<mrow>
  <munderover>
    <mo>∧</mo>
    <mrow><mi>i</mi><mo>=</mo><mn>0</mn></mrow>
    <mi>n</mi>
  </munderover>
  <mrow>
    <mo>(</mo>
    <msub><mi>a</mi><mi>i</mi></msub>
    <mo>&gt;</mo>
    <mn>0</mn>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\bigwedge_{i=0}^{n} (a_i > 0)$$

4.3.5.6 N-ary Linear Algebra Operators: `<selector/>`

Operator Syntax, Schema Class

The `selector` element is the operator for indexing into vectors, matrices and lists. It accepts one or more arguments. The first argument identifies the vector, matrix or list from which the selection is taking place, and the second and subsequent arguments, if any, indicate the kind of selection taking place.

When `selector` is used with a single argument, it should be interpreted as giving the sequence of all elements in the list, vector or matrix given. The ordering of elements in the sequence for a matrix is understood to be first by column, then by row; so the resulting list is of matrix rows given entry by entry. That is, for a matrix $(a_{i,j})$, where the indices denote row and column, respectively, the ordering would be $a_{1,1}, a_{1,2}, \ldots, a_{2,1}, a_{2,2}, \ldots$ etc.

When two arguments are given, and the first is a vector or list, the second argument specifies the index of an entry in the list or vector. If the first argument is a matrix then the second argument specifies the index of a matrix row.

When three arguments are given, the last one is ignored for a list or vector, and in the case of a matrix, the second and third arguments specify the row and column indices of the selected element.

4.3.5.6.1 Examples

▸ Show Section

Content MathML

```
<apply><selector/><ci type="vector">V</ci><cn>1</cn></apply>
```

```
<apply><eq/>
  <apply><selector/>
    <matrix>
      <matrixrow><cn>1</cn><cn>2</cn></matrixrow>
      <matrixrow><cn>3</cn><cn>4</cn></matrixrow>
    </matrix>
    <cn>1</cn>
  </apply>
  <matrix>
    <matrixrow><cn>1</cn><cn>2</cn></matrixrow>
  </matrix>
</apply>
```

Sample Presentation

```
<msub><mi>V</mi><mn>1</mn></msub>
```

$$V_1$$

```
<mrow>
  <msub>
    <mrow>
      <mo>(</mo>
      <mtable>
        <mtr><mtd><mn>1</mn></mtd><mtd><mn>2</mn></mtd></mtr>
        <mtr><mtd><mn>3</mn></mtd><mtd><mn>4</mn></mtd></mtr>
      </mtable>
      <mo>)</mo>
    </mrow>
    <mn>1</mn>
  </msub>
  <mo>=</mo>
  <mrow>
    <mo>(</mo>
    <mtable><mtr><mtd><mn>1</mn></mtd><mtd><mn>2</mn></mtd></mtr></mtable>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}_1 = \begin{pmatrix} 1 & 2 \end{pmatrix}$$

4.3.5.7 N-ary Set Operators: **<union/>**, **<intersect/>**, **<cartesianproduct/>**

Operator Syntax, Schema Class

The `union` element is used to denote the n-ary union of sets. It takes sets as arguments, and denotes the set that contains all the elements that occur in any of them.

The `intersect` element is used to denote the n-ary union of sets. It takes sets as arguments, and denotes the set that contains all the elements that occur in all of them.

The `cartesianproduct` element is used to represent the Cartesian product operator.

Arguments may be explicitly specified in the enclosing `apply` element, or specified using qualifier elements as described in 4.3.5 N-ary Operators.

4.3.5.7.1 EXAMPLES

▸ Show Section

Content MathML

```
<apply><union/><ci>A</ci><ci>B</ci></apply>
```

```
<apply><intersect/><ci>A</ci><ci>B</ci><ci>C</ci></apply>
```

```
<apply><cartesianproduct/><ci>A</ci><ci>B</ci></apply>
```

Sample Presentation

```
<mrow><mi>A</mi><mo>∪</mo><mi>B</mi></mrow>
```

$$A \cup B$$

```
<mrow><mi>A</mi><mo>∩</mo><mi>B</mi><mo>∩</mo><mi>C</mi></mrow>
```

$$A \cap B \cap C$$

```
<mrow><mi>A</mi><mo>×</mo><mi>B</mi></mrow>
```

$$A \times B$$

§ 4.3.5.7.2 Examples (Qualifiers)

▸ Show Section

Content MathML

```
<apply><union/>
  <bvar><ci type="set">S</ci></bvar>
  <domainofapplication>
    <ci type="list">L</ci>
  </domainofapplication>
  <ci type="set"> S</ci>
</apply>
```

```
<apply><intersect/>
  <bvar><ci type="set">S</ci></bvar>
  <domainofapplication>
    <ci type="list">L</ci>
  </domainofapplication>
  <ci type="set"> S</ci>
</apply>
```

Sample Presentation

```
<mrow><munder><mo>∪</mo><mi>L</mi></munder><mi>S</mi></mrow>
```

$$\bigcup_{L} S$$

```
<mrow><munder><mo>∩</mo><mi>L</mi></munder><mi>S</mi></mrow>
```

$$\bigcap_{L} S$$

4.3.5.8 N-ary Matrix Constructors: **<vector/>**, **<matrix/>**, **<matrixrow/>**

Operator Syntax, Schema Class

A vector is an ordered n-tuple of values representing an element of an n-dimensional vector space.

For purposes of interaction with matrices and matrix multiplication, vectors are regarded as equivalent to a matrix consisting of a single column, and the transpose of a vector as a matrix consisting of a single row.

The components of a `vector` may be given explicitly as child elements, or specified by rule as described in 4.3.1.1 Container Markup for Constructor Symbols.

4.3.5.8.1 EXAMPLES

▸ Show Section

Content MathML

```
<vector>
  <apply><plus/><ci>x</ci><ci>y</ci></apply>
  <cn>3</cn>
  <cn>7</cn>
</vector>
```

Sample Presentation

```
<mrow>
  <mo>(</mo>
  <mtable>
    <mtr><mtd><mrow><mi>x</mi><mo>+</mo><mi>y</mi></mrow></mtd></mtr>
    <mtr><mtd><mn>3</mn></mtd></mtr>
    <mtr><mtd><mn>7</mn></mtd></mtr>
  </mtable>
  <mo>)</mo>
</mrow>
```

$$\begin{pmatrix} x+y \\ 3 \\ 7 \end{pmatrix}$$

```
<mrow>
  <mo>(</mo>
  <mrow><mi>x</mi><mo>+</mo><mi>y</mi></mrow>
  <mo>,</mo>
  <mn>3</mn>
  <mo>,</mo>
  <mn>7</mn>
  <mo>)</mo>
</mrow>
```

$$(x + y, 3, 7)$$

A matrix is regarded as made up of matrix rows, each of which can be thought of as a special type of vector.

Note that the behavior of the `matrix` and `matrixrow` elements is substantially different from the `mtable` and `mtr` presentation elements.

The `matrix` element is a *constructor* element, so the entries may be given explicitly as child elements, or specified by rule as described in 4.3.1.1 Container Markup for Constructor Symbols. In the latter case, the entries are specified by providing a function and a 2-dimensional domain of application. The entries of the matrix correspond to the values obtained by evaluating the function at the points of the domain.

Matrix rows are not directly rendered by themselves outside of the context of a matrix.

§ 4.3.5.8.2 Example

▸ Show Section

Content MathML

```
<matrix>
  <bvar><ci type="integer">i</ci></bvar>
  <bvar><ci type="integer">j</ci></bvar>
  <condition>
    <apply><and/>
      <apply><in/>
        <ci>i</ci>
        <interval><ci>1</ci><ci>5</ci></interval>
      </apply>
      <apply><in/>
        <ci>j</ci>
        <interval><ci>5</ci><ci>9</ci></interval>
      </apply>
    </apply>
  </condition>
  <apply><power/><ci>i</ci><ci>j</ci></apply>
</matrix>
```

Sample Presentation

```
<mrow>
  <mo>[</mo>
  <msub><mi>m</mi><mrow><mi>i</mi><mo>,</mo><mi>j</mi></mrow></msub>
  <mo>|</mo>
  <mrow>
    <msub><mi>m</mi><mrow><mi>i</mi><mo>,</mo><mi>j</mi></mrow></msub>
    <mo>=</mo>
    <msup><mi>i</mi><mi>j</mi></msup>
  </mrow>
  <mo>;</mo>
  <mrow>
    <mrow>
      <mi>i</mi>
      <mo>∈</mo>
      <mrow><mo>[</mo><mi>1</mi><mo>,</mo><mi>5</mi><mo>]</mo></mrow>
    </mrow>
    <mo>∧</mo>
    <mrow>
      <mi>j</mi>
      <mo>∈</mo>
      <mrow><mo>[</mo><mi>5</mi><mo>,</mo><mi>9</mi><mo>]</mo></mrow>
    </mrow>
  </mrow>
  <mo>]</mo>
</mrow>
```

$$\left[m_{i,j} \mid m_{i,j} = i^j; i \in [1,5] \land j \in [5,9]\right]$$

4.3.5.9 N-ary Set Theoretic Constructors: `<set>`, `<list>`

Operator Syntax, Schema Class

The `set` element represents the n-ary function which constructs a mathematical set from its arguments. The `set` element takes the attribute `type` which may have the values `set` and `multiset`. The members of the set to be constructed may be given explicitly as child elements of the constructor, or specified by rule as described in 4.3.1.1 Container Markup for Constructor Symbols. There is no implied ordering to the elements of a set.

The `list` element represents the n-ary function which constructs a list from its arguments. Lists differ from sets in that there is an explicit order to the elements. The `list` element takes the attribute `order` which may have the values `numeric` and `lexicographic`. The list entries and order may be given explicitly or specified by rule as described in 4.3.1.1 Container Markup for Constructor Symbols.

4.3.5.9.1 Examples (Explicit elements)

▸ Show Section

Content MathML

```
<set>
  <ci>a</ci><ci>b</ci><ci>c</ci>
</set>
```

```
<list>
  <ci>a</ci><ci>b</ci><ci>c</ci>
</list>
```

Sample Presentation

```
<mrow>
  <mo>{</mo><mi>a</mi><mo>,</mo><mi>b</mi><mo>,</mo><mi>c</mi><mo>}</mo>
</mrow>
```

$$\{a, b, c\}$$

```
<mrow>
  <mo>(</mo><mi>a</mi><mo>,</mo><mi>b</mi><mo>,</mo><mi>c</mi><mo>)</mo>
</mrow>
```

$$(a, b, c)$$

In general sets and lists can be constructed by providing a function and a domain of application. The elements correspond to the values obtained by evaluating the function at the points of the domain. When this method is used for lists, the ordering of the list elements may not be clear, so the kind of ordering may be specified by the `order` attribute. Two orders are supported: lexicographic and numeric.

4.3.5.9.2 Examples (Elements specified by condition)

▸ Show Section

Content MathML

```
<set>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><lt/><ci>x</ci><cn>5</cn></apply>
  </condition>
  <ci>x</ci>
</set>
```

```
<list order="numeric">
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><lt/><ci>x</ci><cn>5</cn></apply>
  </condition>
</list>
```

```
<set>
  <bvar><ci type="set">S</ci></bvar>
  <condition>
    <apply><in/><ci>S</ci><ci type="list">T</ci></apply>
  </condition>
  <ci>S</ci>
</set>
```

```
<set>
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply><and/>
      <apply><lt/><ci>x</ci><cn>5</cn></apply>
      <apply><in/><ci>x</ci><naturalnumbers/></apply>
    </apply>
  </condition>
  <ci>x</ci>
</set>
```

Sample Presentation

```
<mrow>
  <mo>{</mo>
  <mi>x</mi>
  <mo>|</mo>
  <mrow><mi>x</mi><mo>&lt;</mo><mn>5</mn></mrow>
  <mo>}</mo>
</mrow>
```

$$\{x \mid x < 5\}$$

```
<mrow>
  <mo>(</mo>
  <mi>x</mi>
  <mo>|</mo>
  <mrow><mi>x</mi><mo>&lt;</mo><mn>5</mn></mrow>
  <mo>)</mo>
</mrow>
```

$$(x \mid x < 5)$$

```
<mrow>
  <mo>{</mo>
  <mi>S</mi>
  <mo>|</mo>
  <mrow><mi>S</mi><mo>∈</mo><mi>T</mi></mrow>
  <mo>}</mo>
</mrow>
```

$$\{S \mid S \in T\}$$

```
<mrow>
  <mo>{</mo>
  <mi>x</mi>
  <mo>|</mo>
  <mrow>
    <mrow><mo>(</mo><mi>x</mi><mo>&lt;</mo><mn>5</mn><mo>)</mo></mrow>
    <mo>∧</mo>
    <mrow>
      <mi>x</mi><mo>∈</mo><mi mathvariant="double-struck">N</mi>
    </mrow>
  </mrow>
  <mo>}</mo>
</mrow>
```

$$\{x \mid (x < 5) \land x \in \mathbb{N}\}$$

4.3.5.10 N-ary Arithmetic Relations: **<eq/>**, **<gt/>**, **<lt/>**, **<geq/>**, **<leq/>**

Operator Syntax, Schema Class

MathML allows transitive relations to be used with multiple arguments, to give a natural expression to "chains" of relations such as $a < b < c < d$. However unlike the case of the arithmetic operators, the underlying symbols used in the Strict Content MathML are classed as binary, so it is not possible to use apply_to_list as in the previous section, but instead a similar function predicate_on_list is used, the semantics of which is essentially to take the conjunction of applying the predicate to elements of the domain two at a time.

The elements `eq`, `gt`, `lt`, `geq`, `leq` represent respectively the "equality", "greater than", "less than", "greater than or equal to" and "less than or equal to" relations that return true or false depending on the relationship of the first argument to the second.

4.3.5.10.1 Examples

▸ Show Section

Content MathML

```
<apply><eq/>
   <ci>x</ci>
   <cn type="rational">2<sep/>4</cn>
   <cn type="rational">1<sep/>2</cn>
 </apply>
```

```
<apply><gt/><ci>x</ci><ci>y</ci></apply>
```

```
<apply><lt/><ci>y</ci><ci>x</ci></apply>
```

```
<apply><geq/><cn>4</cn><cn>3</cn><cn>3</cn></apply>
```

```
<apply><leq/><cn>3</cn><cn>3</cn><cn>4</cn></apply>
```

Sample Presentation

```
<mrow>
  <mi>x</mi>
  <mo>=</mo>
  <mrow><mn>2</mn><mo>/</mo><mn>4</mn></mrow>
  <mo>=</mo>
  <mrow><mn>1</mn><mo>/</mo><mn>2</mn></mrow>
</mrow>
```

$$x = 2 / 4 = 1 / 2$$

```
<mrow><mi>x</mi><mo>&gt;</mo><mi>y</mi></mrow>
```

$$x > y$$

```
<mrow><mi>y</mi><mo>&lt;</mo><mi>x</mi></mrow>
```

$$y < x$$

```
<mrow><mn>4</mn><mo>≥</mo><mn>3</mn><mo>≥</mo><mn>3</mn></mrow>
```

$$4 \geq 3 \geq 3$$

```
<mrow><mn>3</mn><mo>≤</mo><mn>3</mn><mo>≤</mo><mn>4</mn></mrow>
```

$$3 \leq 3 \leq 4$$

4.3.5.11 N-ary Set Theoretic Relations: **<subset/>**, **<prsubset/>**

Operator Syntax, Schema Class

MathML allows transitive relations to be used with multiple arguments, to give a natural expression to "chains" of relations such as $a < b < c < d$. However unlike the case of the arithmetic operators, the underlying symbols used in the Strict Content MathML are classed as binary, so it is not possible to use apply_to_list as in the previous section, but instead a similar function predicate_on_list is used, the semantics of which is essentially to take the conjunction of applying the predicate to elements of the domain two at a time.

The `subset` and `prsubset` elements represent respectively the subset and proper subset relations. They are used to denote that the first argument is a subset or proper subset of the second. As described above they may also be used as an n-ary operator to express that each argument is a subset or proper subset of its predecessor.

4.3.5.11.1 Examples

▸ Show Section

Content MathML

```
<apply><subset/>
  <ci type="set">A</ci>
  <ci type="set">B</ci>
</apply>
```

```
<apply><prsubset/>
  <ci type="set">A</ci>
  <ci type="set">B</ci>
  <ci type="set">C</ci>
</apply>
```

Sample Presentation

```
<mrow><mi>A</mi><mo>⊆</mo><mi>B</mi></mrow>
```

$$A \subseteq B$$

```
<mrow><mi>A</mi><mo>⊂</mo><mi>B</mi><mo>⊂</mo><mi>C</mi></mrow>
```

$$A \subset B \subset C$$

4.3.5.12 N-ary/Unary Arithmetic Operators: <min/>, <max/>

Operator Syntax, Schema Class

The MathML elements max, min and some statistical elements such as mean may be used as an n-ary function as in the above classes, however a special interpretation is given in the case that a single argument is supplied. If a single argument is supplied the function is applied to the elements represented by the argument.

The underlying symbol used in Strict Content MathML for these elements is *Unary* and so if the MathML is used with 0 or more than 1 argument, the function is applied to the set constructed from the explicitly supplied arguments according to the following rule.

The min element denotes the minimum function, which returns the smallest of the arguments to which it is applied. Its arguments may be explicitly specified in the enclosing apply element, or specified using qualifier elements as described in 4.3.5.12 N-ary/Unary Arithmetic Operators: <min/>, <max/>. Note that when applied to infinite sets of arguments, no minimal argument may exist.

The max element denotes the maximum function, which returns the largest of the arguments to which it is applied. Its arguments may be explicitly specified in the enclosing apply element, or specified using qualifier elements as described in 4.3.5.12 N-ary/Unary Arithmetic Operators: <min/>, <max/>. Note that when applied to infinite sets of arguments, no maximal argument may exist.

4.3.5.12.1 Examples

▸ Show Section

Content MathML

```
<apply><min/><ci>a</ci><ci>b</ci></apply>
```

```
<apply><max/><cn>2</cn><cn>3</cn><cn>5</cn></apply>
```

```
<apply><min/>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><notin/><ci>x</ci><ci type="set">B</ci></apply>
  </condition>
  <apply><power/><ci>x</ci><cn>2</cn></apply>
</apply>
```

```
<apply><max/>
  <bvar><ci>y</ci></bvar>
  <condition>
    <apply><in/>
      <ci>y</ci>
      <interval><cn>0</cn><cn>1</cn></interval>
    </apply>
  </condition>
  <apply><power/><ci>y</ci><cn>3</cn></apply>
</apply>
```

Sample Presentation

```
<mrow>
  <mi>min</mi>
  <mrow><mo>{</mo><mi>a</mi><mo>,</mo><mi>b</mi><mo>}</mo></mrow>
</mrow>
```

$$\min\{a, b\}$$

```
<mrow>
  <mi>max</mi>
  <mrow>
    <mo>{</mo><mn>2</mn><mo>,</mo><mn>3</mn><mo>,</mo><mn>5</mn><mo>}</mo>
  </mrow>
</mrow>
```

$$\max\{2, 3, 5\}$$

```
<mrow>
  <mi>min</mi>
  <mrow><mo>{</mo><msup><mi>x</mi><mn>2</mn></msup><mo>|</mo>
    <mrow><mi>x</mi><mo>∉</mo><mi>B</mi></mrow>
    <mo>}</mo>
  </mrow>
</mrow>
```

$$\min\{x^2 \mid x \notin B\}$$

```
<mrow>
  <mi>max</mi>
  <mrow>
    <mo>{</mo><mi>y</mi><mo>|</mo>
    <mrow>
      <msup><mi>y</mi><mn>3</mn></msup>
      <mo>∈</mo>
      <mrow><mo>[</mo><mn>0</mn><mo>,</mo><mn>1</mn><mo>]</mo></mrow>
    </mrow>
    <mo>}</mo>
  </mrow>
</mrow>
```

$$\max\{y \mid y^3 \in [0,1]\}$$

4.3.5.13 N-ary/Unary Statistical Operators: **<mean/>**, **<median/>**, **<mode/>**, **<sdev/>**, **<variance/>**

Operator Syntax, Schema Class

Some statistical MathML elements, elements such as `mean` may be used as an n-ary function as in the above classes, however a special interpretation is given in the case that a single argument is supplied. If a single argument is supplied the function is applied to the elements represented by the argument.

The underlying symbol used in Strict Content MathML for these elements is *Unary* and so if the MathML is used with 0 or more than 1 argument, the function is applied to the set constructed from the explicitly supplied arguments according to the following rule.

The `mean` element represents the function returning arithmetic mean or average of a data set or random variable.

The `median` element represents an operator returning the median of its arguments. The median is a number separating the lower half of the sample values from the upper half.

The `mode` element is used to denote the mode of its arguments. The mode is the value which occurs with the greatest frequency.

The `sdev` element is used to denote the standard deviation function for a data set or random variable. Standard deviation is a statistical measure of dispersion given by the square root of the variance.

The `variance` element represents the variance of a data set or random variable. Variance is a statistical measure of dispersion, averaging the squares of the deviations of possible values from their mean.

4.3.5.13.1 Examples

▸ Show Section

Content MathML

```
<apply><mean/>
  <cn>3</cn><cn>4</cn><cn>3</cn><cn>7</cn><cn>4</cn>
</apply>
```

```
<apply><mean/><ci>X</ci></apply>
```

```
<apply><sdev/>
  <cn>3</cn><cn>4</cn><cn>2</cn><cn>2</cn>
</apply>
```

```
<apply><sdev/>
  <ci type="discrete_random_variable">X</ci>
</apply>
```

```
<apply><variance/>
  <cn>3</cn><cn>4</cn><cn>2</cn><cn>2</cn>
</apply>
```

```
<apply><variance/>
  <ci type="discrete_random_variable">X</ci>
</apply>
```

Sample Presentation

```
<mrow>
  <mo>⟨</mo>
  <mn>3</mn><mo>,</mo><mn>4</mn><mo>,</mo><mn>3</mn>
  <mo>,</mo><mn>7</mn><mo>,</mo><mn>4</mn>
  <mo>⟩</mo>
</mrow>
```

$$\langle 3, 4, 3, 7, 4 \rangle$$

```
<mrow>
  <mo>⟨</mo><mi>X</mi><mo>⟩</mo>
</mrow>
<mtext> or </mtext>
<mover><mi>X</mi><mo>¯</mo></mover>
```

$\langle X \rangle$ or $\overline{X}$

```
<mrow>
  <mo>σ</mo>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
    <mo>(</mo>
    <mn>3</mn><mo>,</mo><mn>4</mn><mo>,</mo><mn>2</mn><mo>,</mo><mn>2</mn>
    <mo>)</mo>
  </mrow>
</mrow>
```

$\sigma\left(3,4,2,2\right)$

```
<mrow>
  <mo>σ</mo>
  <mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>X</mi><mo>)</mo></mrow>
</mrow>
```

$\sigma\left(X\right)$

```
<mrow>
  <msup>
    <mo>σ</mo>
    <mn>2</mn>
  </msup>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
    <mo>(</mo>
    <mn>3</mn><mo>,</mo><mn>4</mn><mo>,</mo><mn>2</mn><mo>,</mo><mn>2</mn>
    <mo>)</mo>
  </mrow>
</mrow>
```

$\sigma^{2}\left(3,4,2,2\right)$

```
<mrow>
  <msup><mo>σ</mo><mn>2</mn></msup>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow><mo>(</mo><mi>X</mi><mo>)</mo></mrow>
</mrow>
```

$$\sigma^2(X)$$

4.3.6 Binary Operators

Binary operators take two arguments and simply map to OpenMath symbols via Rewrite: element without the need of any special rewrite rules. The binary constructor `interval` is similar but uses constructor syntax in which the arguments are children of the element, and the symbol used depends on the type element as described in 4.3.10.3 Interval `<interval>`.

4.3.6.1 Binary Arithmetic Operators: **`<quotient/>`**, **`<divide/>`**, **`<minus/>`**, **`<power/>`**, **`<rem/>`**, **`<root/>`**

Operator Syntax, Schema Class

The `quotient` element represents the integer division operator. When the operator is applied to integer arguments a and b, the result is the "quotient of a divided by b". That is, the quotient of integers a and b is the integer q such that $a = b * q + r$, with $|r|$ less than $|b|$ and $a * r$ positive. In common usage, q is called the quotient and r is the remainder.

The `divide` element represents the division operator in a number field.

The `minus` element can be used as a unary arithmetic operator (e.g. to represent $-x$), or as a binary arithmetic operator (e.g. to represent $x - y$). Some further examples are given in 4.3.7.2 Unary Arithmetic Operators: `<factorial/>`, `<abs/>`, `<conjugate/>`, `<arg/>`, `<real/>`, `<imaginary/>`, `<floor/>`, `<ceiling/>`, `<exp/>`, `<minus/>`, `<root/>`.

The `power` element represents the exponentiation operator. The first argument is raised to the power of the second argument.

The `rem` element represents the modulus operator, which returns the remainder that results from dividing the first argument by the second. That is, when applied to integer arguments a and b, it returns the unique integer r such that $a = b * q + r$, with $|r|$ less than $|b|$ and $a * r$ positive.

The `root` element is used to extract roots. The kind of root to be taken is specified by a "degree" element, which should be given as the second child of the `apply` element enclosing the `root` element. Thus, square roots correspond to the case where `degree` contains the value 2, cube roots correspond to 3, and so on. If no `degree` is present, a default value of 2 is used.

4.3.6.1.1 Examples

▸ Show Section

Content MathML

```
<apply><quotient/><ci>a</ci><ci>b</ci></apply>
```

```
<apply><divide/>
  <ci>a</ci>
  <ci>b</ci>
</apply>
```

```
<apply><minus/><ci>x</ci><ci>y</ci></apply>
```

```
<apply><power/><ci>x</ci><cn>3</cn></apply>
```

```
<apply><rem/><ci> a </ci><ci> b </ci></apply>
```

```
<apply><root/>
  <degree><ci type="integer">n</ci></degree>
  <ci>a</ci>
</apply>
```

Sample Presentation

```
<mrow><mo>[</mo><mi>a</mi><mo>/</mo><mi>b</mi><mo>]</mo></mrow>
```

$$[a / b]$$

```
<mrow><mi>a</mi><mo>/</mo><mi>b</mi></mrow>
```

$$a / b$$

```
<mrow><mi>x</mi><mo>−</mo><mi>y</mi></mrow>
```

$$x - y$$

```
<msup><mi>x</mi><mn>3</mn></msup>
```

$$x^3$$

```
<mrow><mi>a</mi><mo>mod</mo><mi>b</mi></mrow>
```

$a \bmod b$

```
<mroot><mi>a</mi><mi>n</mi></mroot>
```

$\sqrt[n]{a}$

4.3.6.2 Binary Logical Operators: **<implies/>**, **<equivalent/>**

Operator Syntax, Schema Class

The `implies` element represents the logical implication function which takes two Boolean expressions as arguments. It evaluates to false if the first argument is true and the second argument is false, otherwise it evaluates to true.

The `equivalent` element represents the relation that asserts two Boolean expressions are logically equivalent, that is have the same Boolean value for any inputs.

4.3.6.2.1 Examples

▸ Show Section

Content MathML

```
<apply><implies/><ci>A</ci><ci>B</ci></apply>
```

```
<apply><equivalent/>
  <ci>a</ci>
  <apply><not/><apply><not/><ci>a</ci></apply></apply>
</apply>
```

Sample Presentation

```
<mrow><mi>A</mi><mo>⇒</mo><mi>B</mi></mrow>
```

$A \Rightarrow B$

```
<mrow>
  <mi>a</mi>
  <mo>≡</mo>
  <mrow><mo>¬</mo><mrow><mo>¬</mo><mi>a</mi></mrow></mrow>
</mrow>
```

$$a \equiv \neg\neg a$$

4.3.6.3 Binary Relations: **<neq/>**, **<approx/>**, **<factorof/>**, **<tendsto/>**

Operator Syntax, Schema Class

The `neq` element represents the binary inequality relation, i.e. the relation “not equal to” which returns true unless the two arguments are equal.

The `approx` element represents the relation that asserts the approximate equality of its arguments.

The `factorof` element is used to indicate the mathematical relationship that the first argument “is a factor of” the second. This relationship is true if and only if $b \bmod a = 0$.

4.3.6.3.1 EXAMPLES

▸ Show Section

Content MathML

```
<apply><neq/><cn>3</cn><cn>4</cn></apply>
```

```
<apply><approx/>
  <pi/>
  <cn type="rational">22<sep/>7</cn>
</apply>
```

```
<apply><factorof/><ci>a</ci><ci>b</ci></apply>
```

Sample Presentation

```
<mrow><mn>3</mn><mo>≠</mo><mn>4</mn></mrow>
```

$$3 \neq 4$$

```
<mrow>
  <mi>π</mi>
  <mo>≃</mo>
  <mrow><mn>22</mn><mo>/</mo><mn>7</mn></mrow>
</mrow>
```

$$\pi \simeq 22 / 7$$

```
<mrow><mi>a</mi><mo>|</mo><mi>b</mi></mrow>
```

$$a \mid b$$

The `tendsto` element is used to express the relation that a quantity is tending to a specified value. While this is used primarily as part of the statement of a mathematical limit, it exists as a construct on its own to allow one to capture mathematical statements such as "As x tends to y," and to provide a building block to construct more general kinds of limits.

The `tendsto` element takes the attribute `type` to set the direction from which the limiting value is approached. It may have any value, but common values include `above` and `below`.

§ 4.3.6.3.2 Examples

▸ Show Section

Content MathML

```
<apply><tendsto type="above"/>
  <apply><power/><ci>x</ci><cn>2</cn></apply>
  <apply><power/><ci>a</ci><cn>2</cn></apply>
</apply>
```

```
<apply><tendsto/>
  <vector><ci>x</ci><ci>y</ci></vector>
  <vector>
    <apply><ci type="function">f</ci><ci>x</ci><ci>y</ci></apply>
    <apply><ci type="function">g</ci><ci>x</ci><ci>y</ci></apply>
  </vector>
</apply>
```

Sample Presentation

```
<mrow>
  <msup><mi>x</mi><mn>2</mn></msup>
  <mo>→</mo>
  <msup><msup><mi>a</mi><mn>2</mn></msup><mo>+</mo></msup>
</mrow>
```

$$x^2 \to a^{2+}$$

```
<mrow><mo>(</mo><mtable>
  <mtr><mtd><mi>x</mi></mtd></mtr>
  <mtr><mtd><mi>y</mi></mtd></mtr>
</mtable><mo>)</mo></mrow>
<mo>→</mo>
<mrow><mo>(</mo><mtable>
  <mtr><mtd>
    <mi>f</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>x</mi><mo>,</mo><m
  </mtd></mtr>
  <mtr><mtd>
    <mi>g</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>x</mi><mo>,</mo><m
  </mtd></mtr>
</mtable><mo>)</mo></mrow>
```

$$\begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} f(x,y) \\ g(x,y) \end{pmatrix}$$

4.3.6.4 Binary Linear Algebra Operators: **<vectorproduct/>**, **<scalarproduct/>**, **<outerproduct/>**

Operator Syntax, Schema Class

The `vectorproduct` element represents the vector product. It takes two three-dimensional vector arguments and represents as value a three-dimensional vector.

The `scalarproduct` element represents the scalar product function. It takes two vector arguments and returns a scalar value.

The `outerproduct` element represents the outer product function. It takes two vector arguments and returns as value a matrix.

4.3.6.4.1 Examples

▶ Show Section

Content MathML

```
<apply><eq/>
  <apply><vectorproduct/>
    <ci type="vector"> A </ci>
    <ci type="vector"> B </ci>
  </apply>
  <apply><times/>
    <ci>a</ci>
    <ci>b</ci>
    <apply><sin/><ci>θ</ci></apply>
    <ci type="vector"> N </ci>
  </apply>
</apply>
```

```
<apply><eq/>
  <apply><scalarproduct/>
    <ci type="vector">A</ci>
    <ci type="vector">B</ci>
  </apply>
  <apply><times/>
    <ci>a</ci>
    <ci>b</ci>
    <apply><cos/><ci>θ</ci></apply>
  </apply>
</apply>
```

```
<apply><outerproduct/>
  <ci type="vector">A</ci>
  <ci type="vector">B</ci>
</apply>
```

Sample Presentation

```
<mrow>
  <mrow><mi>A</mi><mo>×</mo><mi>B</mi></mrow>
  <mo>=</mo>
  <mrow>
    <mi>a</mi>
    <mo>&#x2062;<!--InvisibleTimes--></mo>
    <mi>b</mi>
    <mo>&#x2062;<!--InvisibleTimes--></mo>
    <mrow><mi>sin</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>θ</mi></mrow>
    <mo>&#x2062;<!--InvisibleTimes--></mo>
    <mi>N</mi>
  </mrow>
</mrow>
```

$$A \times B = ab\sin\theta N$$

```
<mrow>
  <mrow><mi>A</mi><mo>.</mo><mi>B</mi></mrow>
  <mo>=</mo>
  <mrow>
    <mi>a</mi>
    <mo>&#x2062;<!--InvisibleTimes--></mo>
    <mi>b</mi>
    <mo>&#x2062;<!--InvisibleTimes--></mo>
    <mrow><mi>cos</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>θ</mi></mrow>
  </mrow>
</mrow>
```

$$A \,.\, B = ab\cos\theta$$

```
<mrow><mi>A</mi><mo>⊗</mo><mi>B</mi></mrow>
```

$$A \otimes B$$

4.3.6.5 Binary Set Operators: **`<in/>`**, **`<notin/>`**, **`<notsubset/>`**, **`<notprsubset/>`**, **`<setdiff/>`**

Operator Syntax, Schema Class

The `in` element represents the set inclusion relation. It has two arguments, an element and a set. It is used to denote that the element is in the given set.

The `notin` represents the negated set inclusion relation. It has two arguments, an element and a set. It is used to denote that the element is not in the given set.

The `notsubset` element represents the negated subset relation. It is used to denote that the first argument is not a subset of the second.

The `notprsubset` element represents the negated proper subset relation. It is used to denote that the first argument is not a proper subset of the second.

The `setdiff` element represents the set difference operator. It takes two sets as arguments, and denotes the set that contains all the elements that occur in the first set, but not in the second.

4.3.6.5.1 Examples

▸ Show Section

Content MathML

```
<apply><in/><ci>a</ci><ci type="set">A</ci></apply>
```

```
<apply><notin/><ci>a</ci><ci type="set">A</ci></apply>
```

```
<apply><notsubset/>
  <ci type="set">A</ci>
  <ci type="set">B</ci>
</apply>
```

```
<apply><notprsubset/>
  <ci type="set">A</ci>
  <ci type="set">B</ci>
</apply>
```

```
<apply><setdiff/>
  <ci type="set">A</ci>
  <ci type="set">B</ci>
</apply>
```

Sample Presentation

```
<mrow><mi>a</mi><mo>∈</mo><mi>A</mi></mrow>
```

$$a \in A$$

```
<mrow><mi>a</mi><mo>∉</mo><mi>A</mi></mrow>
```

$$a \notin A$$

```
<mrow><mi>A</mi><mo>⊄</mo><mi>B</mi></mrow>
```

$$A \not\subset B$$

```
<mrow><mi>A</mi><mo>⊄</mo><mi>B</mi></mrow>
```

$$A \not\subset B$$

```
<mrow><mi>A</mi><mo>\</mo><mi>B</mi></mrow>
```

$$A \setminus B$$

4.3.7 Unary Operators

Unary operators take a single argument and map to OpenMath symbols via Rewrite: element without the need of any special rewrite rules.

4.3.7.1 Unary Logical Operators: **<not/>**

Operator Syntax, Schema Class

The `not` element represents the logical not function which takes one Boolean argument, and returns the opposite Boolean value.

4.3.7.1.1 EXAMPLE

▸ Show Section

Content MathML

```
<apply><not/><ci>a</ci></apply>
```

Sample Presentation

```
<mrow><mo>¬</mo><mi>a</mi></mrow>
```

$$\neg a$$

4.3.7.2 Unary Arithmetic Operators: **<factorial/>**, **<abs/>**, **<conjugate/>**, **<arg/>**, **<real/>**, **<imaginary/>**, **<floor/>**, **<ceiling/>**, **<exp/>**, **<minus/>**, **<root/>**

Operator Syntax, Schema Class

The `factorial` element represents the unary factorial operator on non-negative integers. The factorial of an integer n is given by $n! = n \times (n-1) \times \cdots \times 1$.

The `abs` element represents the absolute value function. The argument should be numerically valued. When the argument is a complex number, the absolute value is often referred to as the modulus.

The `conjugate` element represents the function defined over the complex numbers which returns the complex conjugate of its argument.

The `arg` element represents the unary function which returns the angular argument of a complex number, namely the angle which a straight line drawn from the number to zero makes with the real line (measured anti-clockwise).

The `real` element represents the unary operator used to construct an expression representing the "real" part of a complex number, that is, the x component in $x + iy$.

The `imaginary` element represents the unary operator used to construct an expression representing the "imaginary" part of a complex number, that is, the y component in $x + iy$.

The `floor` element represents the operation that rounds down (towards negative infinity) to the nearest integer. This function takes one real number as an argument and returns an integer.

The `ceiling` element represents the operation that rounds up (towards positive infinity) to the nearest integer. This function takes one real number as an argument and returns an integer.

The `exp` element represents the exponentiation function associated with the inverse of the ln function. It takes one argument.

The `minus` element can be used as a unary arithmetic operator (e.g. to represent $-x$), or as a binary arithmetic operator (e.g. to represent $x - y$). Some further examples are given in 4.3.6.1 Binary Arithmetic Operators: `<quotient/>`, `<divide/>`, `<minus/>`, `<power/>`, `<rem/>`, `<root/>`.

The `root` element in MathML is treated as a unary element taking an optional `degree` qualifier, however it represents the binary operation of taking an nth root, and is described in 4.3.6.1 Binary Arithmetic Operators: `<quotient/>`, `<divide/>`, `<minus/>`, `<power/>`, `<rem/>`, `<root/>`.

§ 4.3.7.2.1 Examples

▸ Show Section

Content MathML

```
<apply><factorial/><ci>n</ci></apply>
```

```
<apply><abs/><ci>x</ci></apply>
```

```
<apply><conjugate/>
  <apply><plus/>
    <ci>x</ci>
    <apply><times/><cn>ⅈ</cn><ci>y</ci></apply>
  </apply>
</apply>
```

```
<apply><arg/>
  <apply><plus/>
    <ci> x </ci>
    <apply><times/><imaginaryi/><ci>y</ci></apply>
  </apply>
</apply>
```

```
<apply><real/>
  <apply><plus/>
    <ci>x</ci>
    <apply><times/><imaginaryi/><ci>y</ci></apply>
  </apply>
</apply>
```

```
<apply><imaginary/>
  <apply><plus/>
    <ci>x</ci>
    <apply><times/><imaginaryi/><ci>y</ci></apply>
  </apply>
</apply>
```

```
<apply><floor/><ci>a</ci></apply>
```

```
<apply><ceiling/><ci>a</ci></apply>
```

```
<apply><exp/><ci>x</ci></apply>
```

```
<apply><minus/><cn>3</cn></apply>
```

Sample Presentation

```
<mrow><mi>n</mi><mo>!</mo></mrow>
```

$n!$

```
<mrow><mo>|</mo><mi>x</mi><mo>|</mo></mrow>
```

$$|x|$$

```
<mover>
  <mrow>
    <mi>x</mi>
    <mo>+</mo>
    <mrow><mn>&#x2148;</mn><mo>&#x2062;<!--InvisibleTimes--></mo><mi>y</mi></mrow>
  </mrow>
  <mo>&#x00AF;</mo>
</mover>
```

$$\overline{x+\mathrm{i}y}$$

```
<mrow>
  <mi>arg</mi>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
   <mo>(</mo>
    <mrow>
      <mi>x</mi>
      <mo>+</mo>
      <mrow><mi>i</mi><mo>&#x2062;<!--InvisibleTimes--></mo><mi>y</mi></mrow>
    </mrow>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\arg(x+iy)$$

```
<mrow>
  <mo>&#x211C;</mo>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
   <mo>(</mo>
    <mrow>
      <mi>x</mi>
      <mo>+</mo>
      <mrow><mi>i</mi><mo>&#x2062;<!--InvisibleTimes--></mo><mi>y</mi></mrow>
    </mrow>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\mathcal{R}\,(x+iy)$$

```
<mrow>
  <mo>ℜ</mo>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
   <mo>(</mo>
    <mrow>
      <mi>x</mi>
      <mo>+</mo>
      <mrow><mi>i</mi><mo>&#x2062;<!--InvisibleTimes--></mo><mi>y</mi></mrow>
    </mrow>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\Re\,(x+iy)$$

```
<mrow><mo>⌊</mo><mi>a</mi><mo>⌋</mo></mrow>
```

$$\lfloor a \rfloor$$

```
<mrow><mo>⌈</mo><mi>a</mi><mo>⌉</mo></mrow>
```

$$\lceil a \rceil$$

```
<msup><mi>e</mi><mi>x</mi></msup>
```

$$e^{x}$$

```
<mrow><mo>-</mo><mn>3</mn></mrow>
```

$$-3$$

§ *4.3.7.3 Unary Linear Algebra Operators:* **<determinant/>**, **<transpose/>**

Operator Syntax, Schema Class

The `determinant` element is used for the unary function which returns the determinant of its argument, which should be a square matrix.

The `transpose` element represents a unary function that signifies the transpose of the given matrix or vector.

4.3.7.3.1 Examples

▸ Show Section

Content MathML

```
<apply><determinant/>
  <ci type="matrix">A</ci>
</apply>
```

```
<apply><transpose/>
  <ci type="matrix">A</ci>
</apply>
```

Sample Presentation

```
<mrow><mi>det</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>A</mi></mrow>
```

$$\det A$$

```
<msup><mi>A</mi><mi>T</mi></msup>
```

$$A^T$$

4.3.7.4 Unary Functional Operators: **<inverse/>**, **<ident/>**, **<domain/>**, **<codomain/>**, **<image/>**, **<ln/>**,

Operator Syntax, Schema Class

The `inverse` element is applied to a function in order to construct a generic expression for the functional inverse of that function.

The `ident` element represents the identity function. Note that MathML makes no assumption about the domain and codomain of the represented identity function, which depends on the context in which it is used.

The `domain` element represents the domain of the function to which it is applied. The domain is the set of values over which the function is defined.

The `codomain` represents the codomain, or range, of the function to which it is applied. Note that the codomain is not necessarily equal to the image of the function, it is merely required to contain the image.

The `image` element represents the image of the function to which it is applied. The image of a function is the set of values taken by the function. Every point in the image is generated by the function applied to some point of the domain.

The `ln` element represents the natural logarithm function.

The elements may either be applied to arguments, or may appear alone, in which case they represent an abstract operator acting on other functions.

4.3.7.4.1 Examples

▶ Show Section

Content MathML

```
<apply><inverse/><ci>f</ci></apply>
```

```
<apply>
  <apply><inverse/><ci type="matrix">A</ci></apply>
  <ci>a</ci>
</apply>
```

```
<apply><eq/>
  <apply><compose/>
    <ci type="function">f</ci>
    <apply><inverse/>
      <ci type="function">f</ci>
    </apply>
  </apply>
  <ident/>
</apply>
```

```
<apply><eq/>
  <apply><domain/><ci>f</ci></apply>
  <reals/>
</apply>
```

```
<apply><eq/>
  <apply><codomain/><ci>f</ci></apply>
  <rationals/>
</apply>
```

```
<apply><eq/>
  <apply><image/><sin/></apply>
  <interval><cn>-1</cn><cn> 1</cn></interval>
</apply>
```

```
<apply><ln/><ci>a</ci></apply>
```

Sample Presentation

```
<msup><mi>f</mi><mrow><mo>(</mo><mn>-1</mn><mo>)</mo></mrow></msup>
```

$$f^{(-1)}$$

```
<mrow>
  <msup><mi>A</mi><mrow><mo>(</mo><mn>-1</mn><mo>)</mo></mrow></msup>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow><mo>(</mo><mi>a</mi><mo>)</mo></mrow>
</mrow>
```

$$A^{(-1)}(a)$$

```
<mrow>
  <mrow>
    <mi>f</mi>
    <mo>∘</mo>
    <msup><mi>f</mi><mrow><mo>(</mo><mn>-1</mn><mo>)</mo></mrow></msup>
  </mrow>
  <mo>=</mo>
  <mi>id</mi>
</mrow>
```

$$f \circ f^{(-1)} = \mathrm{id}$$

```
<mrow>
  <mrow><mi>domain</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>f</mi><mc
  <mo>=</mo>
  <mi mathvariant="double-struck">R</mi>
</mrow>
```

$$\operatorname{domain}(f) = \mathbb{R}$$

```
<mrow>
  <mrow><mi>codomain</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>f</mi><
  <mo>=</mo>
  <mi mathvariant="double-struck">Q</mi>
</mrow>
```

$$\operatorname{codomain}(f) = \mathbb{Q}$$

```
<mrow>
  <mrow><mi>image</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>sin</mi><m
  <mo>=</mo>
  <mrow><mo>[</mo><mn>-1</mn><mo>,</mo><mn>1</mn><mo>]</mo></mrow>
</mrow>
```

$$\operatorname{image}(\sin) = [-1, 1]$$

```
<mrow><mi>ln</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>a</mi></mrow>
```

$$\ln a$$

4.3.7.5 Unary Set Operators: `<card/>`

Operator Syntax, Schema Class

The `card` element represents the cardinality function, which takes a set argument and returns its cardinality, i.e. the number of elements in the set. The cardinality of a set is a non-negative integer, or an infinite cardinal number.

4.3.7.5.1 Example

▸ Show Section

Content MathML

```
<apply><eq/>
  <apply><card/><ci>A</ci></apply>
  <cn>5</cn>
</apply>
```

Sample Presentation

```
<mrow>
  <mrow><mo>|</mo><mi>A</mi><mo>|</mo></mrow>
  <mo>=</mo>
  <mn>5</mn>
</mrow>
```

$$|A| = 5$$

4.3.7.6 Unary Elementary Operators: **<sin/>**, **<cos/>**, **<tan/>**, **<sec/>**, **<csc/>**, **<cot/>**, **<sinh/>**, **<cosh/>**, **<tanh/>**, **<sech/>**, **<csch/>**, **<coth/>**, **<arcsin/>**, **<arccos/>**, **<arctan/>**, **<arccosh/>**, **<arccot/>**, **<arccoth/>**, **<arccsc/>**, **<arccsch/>**, **<arcsec/>**, **<arcsech/>**, **<arcsinh/>**, **<arctanh/>**

Operator Syntax, Schema Class

These operator elements denote the standard trigonometric and hyperbolic functions and their inverses. Since their standard interpretations are widely known, they are discussed as a group.

Differing definitions are in use for the inverse functions, so for maximum interoperability applications evaluating such expressions should follow the definitions in [DLMF], Chapter 4: Elementary Functions.

4.3.7.6.1 EXAMPLES

▸ Show Section

Content MathML

```
<apply><sin/><ci>x</ci></apply>
```

```
<apply><sin/>
  <apply><plus/>
    <apply><cos/><ci>x</ci></apply>
    <apply><power/><ci>x</ci><cn>3</cn></apply>
  </apply>
</apply>
```

```
<apply><arcsin/><ci>x</ci></apply>
```

```
<apply><sinh/><ci>x</ci></apply>
```

```
<apply><arcsinh/><ci>x</ci></apply>
```

Sample Presentation

```
<mrow><mi>sin</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi></mrow>
```

$$\sin x$$

```
<mrow>
  <mi>sin</mi>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
    <mo>(</mo>
    <mrow><mi>cos</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi></mrow>
    <mo>+</mo>
    <msup><mi>x</mi><mn>3</mn></msup>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\sin\left(\cos x + x^3\right)$$

```
<mrow>
  <mi>arcsin</mi>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mi>x</mi>
</mrow>
<mtext>  or  </mtext>
<mrow>
  <msup><mi>sin</mi><mrow><mo>-</mo><mn>1</mn></mrow></msup>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mi>x</mi>
</mrow>
```

$$\arcsin x \text{ or } \sin^{-1} x$$

```
<mrow><mi>sinh</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi></mrow>
```

$$\sinh x$$

```
<mrow>
  <mi>arcsinh</mi>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mi>x</mi>
</mrow>
<mtext>  or  </mtext>
<mrow>
  <msup><mi>sinh</mi><mrow><mo>-</mo><mn>1</mn></mrow></msup>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mi>x</mi>
</mrow>
```

$$\operatorname{arcsinh} x \text{ or } \sinh^{-1} x$$

4.3.7.7 Unary Vector Calculus Operators: **<divergence/>**, **<grad/>**, **<curl/>**, **<laplacian/>**

Operator Syntax, Schema Class

The `divergence` element is the vector calculus divergence operator, often called div. It represents the divergence function which takes one argument which should be a vector of scalar-valued functions, intended to represent a vector-valued function, and returns the scalar-valued function giving the divergence of the argument.

The `grad` element is the vector calculus gradient operator, often called grad. It is used to represent the grad function, which takes one argument which should be a scalar-valued function and returns a vector of functions.

The `curl` element is used to represent the curl function of vector calculus. It takes one argument which should be a vector of scalar-valued functions, intended to represent a vector-valued function, and returns a vector of functions.

The `laplacian` element represents the Laplacian operator of vector calculus. The Laplacian takes a single argument which is a vector of scalar-valued functions representing a vector-valued function, and returns a vector of functions.

4.3.7.7.1 Examples

▶ Show Section

Content MathML

```
<apply><divergence/><ci>a</ci></apply>
```

```
<apply><divergence/>
  <ci type="vector">E</ci>
</apply>
```

```
<apply><grad/><ci type="function">f</ci></apply>
```

```
<apply><curl/><ci>a</ci></apply>
```

```
<apply><laplacian/><ci type="vector">E</ci></apply>
```

Sample Presentation

```
<mrow><mi>div</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>a</mi><mo>)</m
```

$$\operatorname{div}(a)$$

```
<mrow><mi>div</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>E</mi><mo>)</m
<mtext> or </mtext>
<mrow><mo>∇</mo><mo>⋅</mo><mi>E</mi></mrow>
```

$$\operatorname{div}(E) \text{ or } \nabla \cdot E$$

```
<mrow>
  <mi>grad</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>f</mi><mo>)</mo><
</mrow>
<mtext> or </mtext>
<mrow>
  <mo>∇</mo><mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow><mo>(</mo><mi>f</mi><mo>)</mo></mrow>
</mrow>
```

$$\mathrm{grad}\,(f) \text{ or } \nabla(f)$$

```
<mrow><mi>curl</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>a</mi><mo>)</
<mtext> or </mtext>
<mrow><mo>∇</mo><mo>×</mo><mi>a</mi></mrow>
```

$$\mathrm{curl}\,(a) \text{ or } \nabla \times a$$

```
<mrow>
  <msup><mo>∇</mo><mn>2</mn></msup>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow><mo>(</mo><mi>E</mi><mo>)</mo></mrow>
</mrow>
```

$$\nabla^2(E)$$

The functions defining the coordinates may be defined implicitly as expressions defined in terms of the coordinate names, in which case the coordinate names must be provided as bound variables.

4.3.7.7.2 Examples

▸ Show Section

Content MathML

```
<apply><divergence/>
  <bvar><ci>x</ci></bvar>
  <bvar><ci>y</ci></bvar>
  <bvar><ci>z</ci></bvar>
  <vector>
    <apply><plus/><ci>x</ci><ci>y</ci></apply>
    <apply><plus/><ci>x</ci><ci>z</ci></apply>
    <apply><plus/><ci>z</ci><ci>y</ci></apply>
  </vector>
</apply>
```

```
<apply><grad/>
  <bvar><ci>x</ci></bvar>
  <bvar><ci>y</ci></bvar>
  <bvar><ci>z</ci></bvar>
  <apply><times/><ci>x</ci><ci>y</ci><ci>z</ci></apply>
</apply>
```

```
<apply><laplacian/>
  <bvar><ci>x</ci></bvar>
  <bvar><ci>y</ci></bvar>
  <bvar><ci>z</ci></bvar>
  <apply><ci>f</ci><ci>x</ci><ci>y</ci></apply>
</apply>
```

Sample Presentation

```
<mrow>
  <mi>div</mi>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mo>(</mo>
  <mtable>
    <mtr><mtd>
      <mi>x</mi>
      <mo>↦</mo>
      <mrow><mi>x</mi><mo>+</mo><mi>y</mi></mrow>
    </mtd></mtr>
    <mtr><mtd>
      <mi>y</mi>
      <mo>↦</mo>
      <mrow><mi>x</mi><mo>+</mo><mi>z</mi></mrow>
    </mtd></mtr>
    <mtr><mtd>
      <mi>z</mi>
      <mo>↦</mo>
      <mrow><mi>z</mi><mo>+</mo><mi>y</mi></mrow>
    </mtd></mtr>
  </mtable>
  <mo>)</mo>
</mrow>
```

$$\operatorname{div}\begin{pmatrix} x \mapsto x+y \\ y \mapsto x+z \\ z \mapsto z+y \end{pmatrix}$$

```
<mrow>
  <mi>grad</mi>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
    <mo>(</mo>
    <mrow><mo>(</mo><mi>x</mi><mo>,</mo><mi>y</mi><mo>,</mo><mi>z</mi><mo>)</mo></mrow>
    <mo>↦</mo>
    <mrow>
      <mi>x</mi><mo>&#x2062;<!--InvisibleTimes--></mo><mi>y</mi><mo>&#x2062;<!--Invisible
    </mrow>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\operatorname{grad}\left((x, y, z) \mapsto xyz\right)$$

```
<mrow>
  <msup><mo>∇</mo><mn>2</mn></msup>
  <mo>&#x2061;<!--ApplyFunction--></mo>
  <mrow>
    <mo>(</mo>
    <mrow><mo>(</mo><mi>x</mi><mo>,</mo><mi>y</mi><mo>,</mo><mi>z</mi><mo>)</mo></mrow>
    <mo>↦</mo>
    <mrow>
      <mi>f</mi>
      <mo>&#x2061;<!--ApplyFunction--></mo>
      <mrow><mo>(</mo><mi>x</mi><mo>,</mo><mi>y</mi><mo>)</mo></mrow>
    </mrow>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\nabla^2((x, y, z) \mapsto f(x, y))$$

4.3.7.8 Moment **<moment/>**, **<momentabout>**

Operator Syntax, Schema Class

The `moment` element is used to denote the *i*th moment of a set of data set or random variable. The `moment` function accepts the `degree` and `momentabout` qualifiers. If present, the `degree` schema denotes the order of the moment. Otherwise, the moment is assumed to be the first order moment. When used with `moment`, the `degree` schema is expected to contain a single child. If present, the `momentabout` schema denotes the point about which the moment is taken. Otherwise, the moment is assumed to be the moment about zero.

4.3.7.8.1 Examples

▸ Show Section

Content MathML

```
<apply><moment/>
  <degree><cn>3</cn></degree>
  <momentabout><mean/></momentabout>
  <cn>6</cn><cn>4</cn><cn>2</cn><cn>2</cn><cn>5</cn>
</apply>
```

```
<apply><moment/>
  <degree><cn>3</cn></degree>
  <momentabout><ci>p</ci></momentabout>
  <ci>X</ci>
</apply>
```

Sample Presentation

```
<msub>
  <mrow>
    <mo>⟨</mo>
    <msup>
      <mrow>
        <mo>(</mo>
        <mn>6</mn><mo>,</mo>
        <mn>4</mn><mo>,</mo>
        <mn>2</mn><mo>,</mo>
        <mn>2</mn><mo>,</mo>
        <mn>5</mn>
        <mo>)</mo>
      </mrow>
      <mn>3</mn>
    </msup>
    <mo>⟩</mo>
  </mrow>
  <mi>mean</mi>
</msub>
```

$$\left\langle (6,4,2,2,5)^{3} \right\rangle_{\text{mean}}$$

```
<msub>
  <mrow>
    <mo>⟨</mo>
    <msup><mi>X</mi><mn>3</mn></msup>
    <mo>⟩</mo>
  </mrow>
  <mi>p</mi>
</msub>
```

$$\left\langle X^{3} \right\rangle_{p}$$

§ *4.3.7.9 Logarithm* **<log/>** , **<logbase>**

Operator Syntax, Schema Class

The `log` element represents the logarithm function relative to a given base. When present, the `logbase` qualifier specifies the base. Otherwise, the base is assumed to be 10.

4.3.7.9.1 Examples

▶ Show Section

Content MathML

```
<apply><log/>
  <logbase><cn>3</cn></logbase>
  <ci>x</ci>
</apply>
```

```
<apply><log/><ci>x</ci></apply>
```

Sample Presentation

```
<mrow><msub><mi>log</mi><mn>3</mn></msub><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi><
```

$$\log_3 x$$

```
<mrow><mi>log</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi></mrow>
```

$$\log x$$

4.3.8 Unary Qualified Calculus Operators

4.3.8.1 Integral **`<int/>`**

Operator Syntax, Schema Class

The `int` element is the operator element for a definite or indefinite integral over a function or a definite integral over an expression with a bound variable.

4.3.8.1.1 Examples

▸ Show Section

Content MathML

```
<apply><eq/>
  <apply><int/><sin/></apply>
  <cos/>
</apply>
```

```
<apply><int/>
  <interval><ci>a</ci><ci>b</ci></interval>
  <cos/>
</apply>
```

Sample Presentation

```
<mrow><mrow><mi>∫</mi><mi>sin</mi></mrow><mo>=</mo><mi>cos</mi></mrow>
```

$$\int \sin = \cos$$

```
<mrow>
  <msubsup><mi>∫</mi><mi>a</mi><mi>b</mi></msubsup><mi>cos</mi>
</mrow>
```

$$\int_{a}^{b} \cos$$

The `int` element can also be used with bound variables serving as the integration variables.

Definite integrals are indicated by providing qualifier elements specifying a domain of integration. A `lowlimit`/`uplimit` pair is perhaps the most “standard” representation of a definite integral.

4.3.8.1.2 Example

▸ Show Section

Content MathML

```
<apply><int/>
  <bvar><ci>x</ci></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <uplimit><cn>1</cn></uplimit>
  <apply><power/><ci>x</ci><cn>2</cn></apply>
</apply>
```

Sample Presentation

```
<mrow>
  <msubsup><mi>∫</mi><mn>0</mn><mn>1</mn></msubsup>
  <msup><mi>x</mi><mn>2</mn></msup>
  <mi>d</mi>
  <mi>x</mi>
</mrow>
```

$$\int_0^1 x^2 dx$$

4.3.8.2 Differentiation **<diff/>**

Operator Syntax, Schema Class

The `diff` element is the differentiation operator element for functions or expressions of a single variable. It may be applied directly to an actual function thereby denoting a function which is the derivative of the original function, or it can be applied to an expression involving a single variable.

4.3.8.2.1 EXAMPLES

▸ Show Section

Content MathML

```
<apply><diff/><ci>f</ci></apply>
```

```
<apply><eq/>
  <apply><diff/>
    <bvar><ci>x</ci></bvar>
    <apply><sin/><ci>x</ci></apply>
  </apply>
  <apply><cos/><ci>x</ci></apply>
</apply>
```

Sample Presentation

```
<msup><mi>f</mi><mo>′</mo></msup>
```

$$f'$$

```
<mrow>
  <mfrac>
    <mrow><mi>d</mi><mrow><mi>sin</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi></mr
    <mrow><mi>d</mi><mi>x</mi></mrow>
  </mfrac>
  <mo>=</mo>
  <mrow><mi>cos</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi></mrow>
</mrow>
```

$$\frac{d\sin x}{dx} = \cos x$$

The `bvar` element may also contain a `degree` element, which specifies the order of the derivative to be taken.

4.3.8.2.2 EXAMPLE

▸ Show Section

Content MathML

```
<apply><diff/>
  <bvar><ci>x</ci><degree><cn>2</cn></degree></bvar>
  <apply><power/><ci>x</ci><cn>4</cn></apply>
</apply>
```

Sample Presentation

```
<mfrac>
  <mrow>
    <msup><mi>d</mi><mn>2</mn></msup>
    <msup><mi>x</mi><mn>4</mn></msup>
  </mrow>
  <mrow><mi>d</mi><msup><mi>x</mi><mn>2</mn></msup></mrow>
</mfrac>
```

$$\frac{d^2x^4}{dx^2}$$

4.3.8.3 Partial Differentiation **<partialdiff/>**

Operator Syntax, Schema Class

The `partialdiff` element is the partial differentiation operator element for functions or expressions in several variables.

For the case of partial differentiation of a function, the containing `partialdiff` takes two arguments: firstly a list of indices indicating by position which function arguments are involved in constructing the partial derivatives, and secondly the actual function to be partially differentiated. The indices may be repeated.

4.3.8.3.1 EXAMPLES

▸ Show Section

Content MathML

```
<apply><partialdiff/>
  <list><cn>1</cn><cn>1</cn><cn>3</cn></list>
  <ci type="function">f</ci>
</apply>
```

```
<apply><partialdiff/>
  <list><cn>1</cn><cn>1</cn><cn>3</cn></list>
  <lambda>
    <bvar><ci>x</ci></bvar>
    <bvar><ci>y</ci></bvar>
    <bvar><ci>z</ci></bvar>
    <apply><ci>f</ci><ci>x</ci><ci>y</ci><ci>z</ci></apply>
  </lambda>
</apply>
```

Sample Presentation

```
<mrow>
  <msub>
    <mi>D</mi>
    <mrow><mn>1</mn><mo>,</mo><mn>1</mn><mo>,</mo><mn>3</mn></mrow>
  </msub>
  <mi>f</mi>
</mrow>
```

$$D_{1,1,3}f$$

```
<mfrac>
  <mrow>
    <msup><mo>∂</mo><mn>3</mn></msup>
    <mrow>
      <mi>f</mi>
      <mo>&#x2061;<!--ApplyFunction--></mo>
      <mrow><mo>(</mo><mi>x</mi><mo>,</mo><mi>y</mi><mo>,</mo><mi>z</mi><mo>)</mo></mrow>
    </mrow>
  </mrow>
  <mrow>
    <mrow><mo>∂</mo><msup><mi>x</mi><mn>2</mn></msup></mrow>
    <mrow><mo>∂</mo><mi>z</mi></mrow>
  </mrow>
</mfrac>
```

$$\frac{\partial^3 f(x,y,z)}{\partial x^2\,\partial z}$$

In the case of algebraic expressions, the bound variables are given by `bvar` elements, which are children of the containing `apply` element. The `bvar` elements may also contain `degree` elements, which specify the order of the partial derivative to be taken in that variable.

Where a total degree of differentiation must be specified, this is indicated by use of a `degree` element at the top level, i.e. without any associated `bvar`, as a child of the containing `apply` element.

§ 4.3.8.3.2 Examples

▸ Show Section

Content MathML

```
<apply><partialdiff/>
  <bvar><ci>x</ci></bvar>
  <bvar><ci>y</ci></bvar>
  <apply><ci type="function">f</ci><ci>x</ci><ci>y</ci></apply>
</apply>
```

```
<apply><partialdiff/>
  <bvar><ci>x</ci><degree><ci>m</ci></degree></bvar>
  <bvar><ci>y</ci><degree><ci>n</ci></degree></bvar>
  <degree><ci>k</ci></degree>
  <apply><ci type="function">f</ci>
    <ci>x</ci>
    <ci>y</ci>
  </apply>
</apply>
```

Sample Presentation

```
<mfrac>
  <mrow>
    <msup><mo>∂</mo><mn>2</mn></msup>
    <mrow>
      <mi>f</mi>
      <mo>&#x2061;<!--ApplyFunction--></mo>
      <mrow><mo>(</mo><mi>x</mi><mo>,</mo><mi>y</mi><mo>)</mo></mrow>
    </mrow>
  </mrow>
  <mrow>
    <mrow><mo>∂</mo><mi>x</mi></mrow>
    <mrow><mo>∂</mo><mi>y</mi></mrow>
  </mrow>
</mfrac>
```

$$\frac{\partial^2 f(x,y)}{\partial x\,\partial y}$$

```
<mfrac>
  <mrow>
    <msup><mo>∂</mo><mi>k</mi></msup>
    <mrow>
      <mi>f</mi>
      <mo>&#x2061;<!--ApplyFunction--></mo>
      <mrow><mo>(</mo><mi>x</mi><mo>,</mo><mi>y</mi><mo>)</mo></mrow>
    </mrow>
  </mrow>
  <mrow>
    <mrow><mo>∂</mo><msup><mi>x</mi><mi>m</mi></msup>
    </mrow>
    <mrow><mo>∂</mo><msup><mi>y</mi><mi>n</mi></msup></mrow>
  </mrow>
</mfrac>
```

$$\frac{\partial^k f(x,y)}{\partial x^m \partial y^n}$$

4.3.9 Constants

Constant symbols relate to mathematical constants such as e and true and also to names of sets such as the Real Numbers, and Integers. In Strict Content MathML, they rewrite simply to the corresponding symbol listed in the syntax tables for Arithmetic Constants and Set Theory Constants.

4.3.9.1 Arithmetic Constants: **<exponentiale/>**, **<imaginaryi/>**, **<notanumber/>**, **<true/>**, **<false/>**, **<pi/>**, **<eulergamma/>**, **<infinity/>**

Operator Syntax, Schema Class

The elements <exponentiale/>, <imaginaryi/>, <notanumber/>, <true/>, <false/>, <pi/>, <eulergamma/>, <infinity/> represent respectively:
the base of the natural logarithm, approximately 2.718;
the square root of -1, commonly written *i*;
not-a-number, i.e. the result of an ill-posed floating point computation (see [IEEE754]);
the Boolean value true;
the Boolean value false;
pi (π), approximately 3.142, which is the ratio of the circumference of a circle to its diameter;
the gamma constant (γ), approximately 0.5772;
infinity (∞).

4.3.9.1.1 Examples

▶ Show Section

Content MathML

```
<apply><eq/><apply><ln/><exponentiale/></apply><cn>1</cn></apply>
```

```
<apply><eq/><apply><power/><imaginaryi/><cn>2</cn></apply><cn>-1</cn></apply>
```

```
<apply><eq/><apply><divide/><cn>0</cn><cn>0</cn></apply><notanumber/></apply>
```

```
<apply><eq/><apply><or/><true/><ci type="boolean">P</ci></apply><true/></apply>
```

```
<apply><eq/><apply><and/><false/><ci type="boolean">P</ci></apply><false/></apply>
```

```
<apply><approx/><pi/><cn type="rational">22<sep/>7</cn></apply>
```

```
<apply><approx/><eulergamma/><cn>0.5772156649</cn></apply>
```

```
<infinity/>
```

Sample Presentation

```
<mrow>
  <mrow><mi>ln</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>e</mi></mrow>
  <mo>=</mo>
  <mn>1</mn>
</mrow>
```

$$\ln e = 1$$

```
<mrow><msup><mi>i</mi><mn>2</mn></msup><mo>=</mo><mn>-1</mn></mrow>
```

$$i^2 = \text{-}1$$

```
<mrow>
  <mrow><mn>0</mn><mo>/</mo><mn>0</mn></mrow>
  <mo>=</mo>
  <mi>NaN</mi>
</mrow>
```

$$0 / 0 = \mathrm{NaN}$$

```
<mrow>
  <mrow><mi>true</mi><mo>∨</mo><mi>P</mi></mrow>
  <mo>=</mo>
  <mi>true</mi>
</mrow>
```

$$\mathrm{true} \vee P = \mathrm{true}$$

```
<mrow>
  <mrow><mi>false</mi><mo>∧</mo><mi>P</mi></mrow>
  <mo>=</mo>
  <mi>false</mi>
</mrow>
```

$$\mathrm{false} \wedge P = \mathrm{false}$$

```
<mrow>
  <mi>π</mi>
  <mo>≃</mo>
  <mrow><mn>22</mn><mo>/</mo><mn>7</mn></mrow>
</mrow>
```

$$\pi \simeq 22 / 7$$

```
<mrow>
  <mi>γ</mi><mo>≃</mo><mn>0.5772156649</mn>
</mrow>
```

$$\gamma \simeq 0.5772156649$$

```
<mi>∞</mi>
```

$$\infty$$

4.3.9.2 Set Theory Constants: **<integers/>**, **<reals/>**, **<rationals/>**, **<naturalnumbers/>**, **<complexes/>**, **<primes/>**, **<emptyset/>**

Operator Syntax, Schema Class

These elements represent the standard number sets, Integers, Reals, Rationals, Natural Numbers (including zero), Complex Numbers, Prime Numbers, and the Empty Set.

4.3.9.2.1 EXAMPLES

▸ Show Section

Content MathML

```
<apply><in/><cn type="integer">42</cn><integers/></apply>
```

```
<apply><in/><cn type="real">44.997</cn><reals/></apply>
```

```
<apply><in/><cn type="rational">22<sep/>7</cn><rationals/></apply>
```

```
<apply><in/><cn type="integer">1729</cn><naturalnumbers/></apply>
```

```
<apply><in/><cn type="complex-cartesian">17<sep/>29</cn><complexes/></apply>
```

```
<apply><in/><cn type="integer">17</cn><primes/></apply>
```

```
<apply><neq/><integers/><emptyset/></apply>
```

Sample Presentation

```
<mrow><mn>42</mn><mo>∈</mo><mi mathvariant="double-struck">Z</mi></mrow>
```

$$42 \in \mathbb{Z}$$

```
<mrow>
  <mn>44.997</mn><mo>∈</mo><mi mathvariant="double-struck">R</mi>
</mrow>
```

$$44.997 \in \mathbb{R}$$

```
<mrow>
  <mrow><mn>22</mn><mo>/</mo><mn>7</mn></mrow>
  <mo>∈</mo>
  <mi mathvariant="double-struck">Q</mi>
</mrow>
```

$$22 / 7 \in \mathbb{Q}$$

```
<mrow>
  <mn>1729</mn><mo>∈</mo><mi mathvariant="double-struck">N</mi>
</mrow>
```

$$1729 \in \mathbb{N}$$

```
<mrow>
  <mrow><mn>17</mn><mo>+</mo><mn>29</mn><mo>&#x2062;<!--InvisibleTimes--></mo><mi>i</mi><
  <mo>∈</mo>
  <mi mathvariant="double-struck">C</mi>
</mrow>
```

$$17 + 29i \in \mathbb{C}$$

```
<mrow><mn>17</mn><mo>∈</mo><mi mathvariant="double-struck">P</mi></mrow>
```

$$17 \in \mathbb{P}$$

```
<mrow>
  <mi mathvariant="double-struck">Z</mi><mo>≠</mo><mi>∅</mi>
</mrow>
```

$$\mathbb{Z} \neq \varnothing$$

4.3.10 Special Element forms

4.3.10.1 Quantifiers: **<forall/>, <exists/>**

Operator Syntax, Schema Class

The `forall` and `<exists/>` elements represent the universal (“for all”) and existential (“there exists”) quantifiers which take one or more bound variables, and an argument which specifies the assertion being quantified. In addition, `condition` or other qualifiers may be used to limit the domain of the bound variables.

4.3.10.1.1 Examples

▸ Show Section

Content MathML

```
<bind><forall/>
  <bvar><ci>x</ci></bvar>
  <apply><eq/>
    <apply><minus/><ci>x</ci><ci>x</ci></apply>
    <cn>0</cn>
  </apply>
</bind>
```

Sample Presentation

```
<mrow>
  <mo>∀</mo>
  <mi>x</mi>
  <mo>.</mo>
  <mrow>
   <mo>(</mo>
    <mrow>
      <mrow><mi>x</mi><mo>−</mo><mi>x</mi></mrow>
      <mo>=</mo>
      <mn>0</mn>
    </mrow>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\forall x\,.\,(x - x = 0)$$

Content MathML

```
<bind><exists/>
  <bvar><ci>x</ci></bvar>
  <apply><eq/>
    <apply><ci>f</ci><ci>x</ci></apply>
    <cn>0</cn>
  </apply>
</bind>
```

Sample Presentation

```
<mrow>
  <mo>∃</mo>
  <mi>x</mi>
  <mo>.</mo>
  <mrow>
   <mo>(</mo>
    <mrow>
      <mrow><mi>f</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>x</mi><mo>
      <mo>=</mo>
      <mn>0</mn>
    </mrow>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\exists x\,.\,(f(x) = 0)$$

Content MathML

```
<apply><exists/>
  <bvar><ci>x</ci></bvar>
  <domainofapplication>
    <integers/>
  </domainofapplication>
  <apply><eq/>
    <apply><ci>f</ci><ci>x</ci></apply>
    <cn>0</cn>
  </apply>
</apply>
```

Strict MathML equivalent:

```
<bind><csymbol cd="quant1">exists</csymbol>
  <bvar><ci>x</ci></bvar>
  <apply><csymbol cd="logic1">and</csymbol>
    <apply><csymbol cd="set1">in</csymbol>
      <ci>x</ci>
      <csymbol cd="setname1">Z</csymbol>
    </apply>
    <apply><csymbol cd="relation1">eq</csymbol>
      <apply><ci>f</ci><ci>x</ci></apply>
      <cn>0</cn>
    </apply>
  </apply>
</bind>
```

Sample Presentation

```
<mrow>
  <mo>∃</mo>
  <mi>x</mi>
  <mo>.</mo>
  <mrow>
   <mo>(</mo>
    <mrow><mi>x</mi><mo>∈</mo><mi mathvariant="double-struck">Z</mi></mrow>
    <mo>∧</mo>
    <mrow>
      <mrow><mi>f</mi><mo>&#x2061;<!--ApplyFunction--></mo><mrow><mo>(</mo><mi>x</mi><mo>
      <mo>=</mo>
      <mn>0</mn>
    </mrow>
    <mo>)</mo>
  </mrow>
</mrow>
```

$$\exists x\,.\,(x \in \mathbb{Z} \land f(x) = 0)$$

4.3.10.2 Lambda `<lambda>`

Operator Syntax, Schema Class

The `lambda` element is used to construct a user-defined function from an expression, bound variables, and qualifiers. In a lambda construct with *n* (possibly 0) bound variables, the first *n* children are `bvar` elements that identify the variables that are used as placeholders in the last child for actual parameter values. The bound variables can be restricted by an optional `domainofapplication` qualifier or one of its shorthand notations. The meaning of the `lambda` construct is an *n*-ary function that returns the expression in the last child where the bound variables are replaced with the respective arguments.

The `domainofapplication` child restricts the possible values of the arguments of the constructed function. For instance,

the following `lambda` construct represents a function on the integers.

```
<lambda>
  <bvar><ci> x </ci></bvar>
  <domainofapplication><integers/></domainofapplication>
  <apply><sin/><ci> x </ci></apply>
</lambda>
```

If a `lambda` construct does not contain bound variables, then the `lambda` construct is superfluous and may be removed, unless it also contains a `domainofapplication` construct. In that case, if the last child of the `lambda` construct is itself a function, then the `domainofapplication` restricts its existing functional arguments, as in this example, which is a variant representation for the function above.

```
<lambda>
  <domainofapplication><integers/></domainofapplication>
  <sin/>
</lambda>
```

Otherwise, if the last child of the `lambda` construct is not a function, say a number, then the `lambda` construct will not be a function, but the same number, and any `domainofapplication` is ignored.

4.3.10.2.1 Examples

▸ Show Section

Content MathML

```
<lambda>
  <bvar><ci>x</ci></bvar>
  <apply><sin/>
    <apply><plus/><ci>x</ci><cn>1</cn></apply>
  </apply>
</lambda>
```

Sample Presentation

```
<mrow>
  <mi>λ</mi>
  <mi>x</mi>
  <mo>.</mo>
  <mrow>
   <mo>(</mo>
    <mrow>
      <mi>sin</mi>
      <mo>&#x2061;<!--ApplyFunction--></mo>
      <mrow><mo>(</mo><mi>x</mi><mo>+</mo><mn>1</mn><mo>)</mo></mrow>
    </mrow>
    <mo>)</mo>
  </mrow>
</mrow>
<mtext> or </mtext>
<mrow>
  <mi>x</mi>
  <mo>↦</mo>
  <mrow>
    <mi>sin</mi>
    <mo>&#x2061;<!--ApplyFunction--></mo>
    <mrow><mo>(</mo><mi>x</mi><mo>+</mo><mn>1</mn><mo>)</mo></mrow>
  </mrow>
</mrow>
```

$$\lambda x\,.\,(\sin\,(x+1))\text{ or }x\mapsto\sin\,(x+1)$$

4.3.10.3 Interval **<interval>**

Operator Syntax, Schema Class

The `interval` element is a container element used to represent simple mathematical intervals of the real number line. It takes an optional attribute `closure`, which can take any of the values `open`, `closed`, `open-closed`, or `closed-open`, with a default value of `closed`.

As described in 4.3.3.1 Uses of <domainofapplication>, <interval>, <condition>, <lowlimit> and <uplimit>, `interval` is interpreted as a qualifier if it immediately follows `bvar`.

4.3.10.3.1 Example

▸ Show Section

Content MathML

```
<interval closure="open"><ci>x</ci><cn>1</cn></interval>
```

```
<interval closure="closed"><cn>0</cn><cn>1</cn></interval>
```

```
<interval closure="open-closed"><cn>0</cn><cn>1</cn></interval>
```

```
<interval closure="closed-open"><cn>0</cn><cn>1</cn></interval>
```

Sample Presentation

```
<mrow><mo>(</mo><mi>x</mi><mo>,</mo><mn>1</mn><mo>)</mo></mrow>
```

$$(x, 1)$$

```
<mrow><mo>[</mo><mn>0</mn><mo>,</mo><mn>1</mn><mo>]</mo></mrow>
```

$$[0, 1]$$

```
<mrow><mo>(</mo><mn>0</mn><mo>,</mo><mn>1</mn><mo>]</mo></mrow>
```

$$(0, 1]$$

```
<mrow><mo>[</mo><mn>0</mn><mo>,</mo><mn>1</mn><mo>)</mo></mrow>
```

$$[0, 1)$$

4.3.10.4 Limits **<limit/>**

Operator Syntax, Schema Class

The `limit` element represents the operation of taking a limit of a sequence. The limit point is expressed by specifying a `lowlimit` and a `bvar`, or by specifying a `condition` on one or more bound variables.

The direction from which a limiting value is approached is given as an argument limit in Strict Content MathML, which supplies the direction specifier symbols both_sides, above, and below for this purpose. The first correspond to the values

`all`, `above`, and `below` of the `type` attribute of the `tendsto` element. The null symbol corresponds to the case where no `type` attribute is present.

4.3.10.4.1 Examples

▸ Show Section

Content MathML

```
<apply><limit/>
  <bvar><ci>x</ci></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <apply><sin/><ci>x</ci></apply>
</apply>
```

```
<apply><limit/>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><tendsto/><ci>x</ci><cn>0</cn></apply>
  </condition>
  <apply><sin/><ci>x</ci></apply>
</apply>
```

```
<apply><limit/>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><tendsto type="above"/><ci>x</ci><ci>a</ci></apply>
  </condition>
  <apply><sin/><ci>x</ci></apply>
</apply>
```

Sample Presentation

```
<mrow>
  <munder>
    <mi>lim</mi>
    <mrow><mi>x</mi><mo>→</mo><mn>0</mn></mrow>
  </munder>
  <mrow><mi>sin</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi></mrow>
</mrow>
```

$$\lim_{x \to 0} \sin x$$

```
<mrow>
  <munder>
    <mi>lim</mi>
    <mrow><mi>x</mi><mo>→</mo><mn>0</mn></mrow>
  </munder>
  <mrow><mi>sin</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi></mrow>
</mrow>
```

$$\lim_{x \to 0} \sin x$$

```
<mrow>
  <munder>
    <mi>lim</mi>
    <mrow><mi>x</mi><mo>→</mo><msup><mi>a</mi><mo>+</mo></msup></mrow>
  </munder>
  <mrow><mi>sin</mi><mo>&#x2061;<!--ApplyFunction--></mo><mi>x</mi></mrow>
</mrow>
```

$$\lim_{x \to a^+} \sin x$$

4.3.10.5 Piecewise declaration **<piecewise>**, **<piece>**, **<otherwise>**

Operator Syntax, Schema Class

The piecewise, piece, and otherwise elements are used to represent "piecewise" function definitions of the form "$H(x) = 0$ if x less than 0, $H(x) = 1$ otherwise".

The declaration is constructed using the piecewise element. This contains zero or more piece elements, and optionally one otherwise element. Each piece element contains exactly two children. The first child defines the value taken by the piecewise expression when the condition specified in the associated second child of the piece is true. The degenerate case of no piece elements and no otherwise element is treated as undefined for all values of the domain.

The otherwise element allows the specification of a value to be taken by the piecewise function when none of the conditions (second child elements of the piece elements) is true, i.e. a default value.

It should be noted that no "order of execution" is implied by the ordering of the piece child elements within piecewise. It is the responsibility of the author to ensure that the subsets of the function domain defined by the second children of the piece elements are disjoint, or that, where they overlap, the values of the corresponding first children of the piece elements coincide. If this is not the case, the meaning of the expression is undefined.

4.3.10.5.1 Example

▸ Show Section

Content MathML

```
<piecewise>
  <piece>
    <apply><minus/><ci>x</ci></apply>
    <apply><lt/><ci>x</ci><cn>0</cn></apply>
  </piece>
  <piece>
    <cn>0</cn>
    <apply><eq/><ci>x</ci><cn>0</cn></apply>
  </piece>
  <piece>
    <ci>x</ci>
    <apply><gt/><ci>x</ci><cn>0</cn></apply>
  </piece>
</piecewise>
```

Sample Presentation

```
<mrow>
  <mo>{</mo>
  <mtable>
    <mtr>
      <mtd><mrow><mo>-</mo><mi>x</mi></mrow></mtd>
      <mtd columnalign="left"><mtext>  if  </mtext></mtd>
      <mtd><mrow><mi>x</mi><mo>&lt;</mo><mn>0</mn></mrow></mtd>
    </mtr>
    <mtr>
      <mtd><mn>0</mn></mtd>
      <mtd columnalign="left"><mtext>  if  </mtext></mtd>
      <mtd><mrow><mi>x</mi><mo>=</mo><mn>0</mn></mrow></mtd>
    </mtr>
    <mtr>
      <mtd><mi>x</mi></mtd>
      <mtd columnalign="left"><mtext>  if  </mtext></mtd>
      <mtd><mrow><mi>x</mi><mo>&gt;</mo><mn>0</mn></mrow></mtd>
    </mtr>
  </mtable>
</mrow>
```

$$\begin{cases} -x & \text{if} \quad x < 0 \\ 0 & \text{if} \quad x = 0 \\ x & \text{if} \quad x > 0 \end{cases}$$

5. Annotating MathML: intent

MathML has been widely adopted by assistive technologies (AT). However, math notations can be ambiguous which can result in AT guessing at what should be spoken in some cases. MathML 4 adds a lightweight method for authors to express their intent: the `intent` attribute. This attribute is similar to the `aria-label` attribute with some important distinctions. In terms of accessibility, the major difference is that `intent` does *not* affect braille generation. Most languages have a separate braille code for math so that the words used for speech should not be affected by braille generation. Some languages, such as English, have more than one braille math code and it is impossible for the author to know which is desired by the reader. Hence, even if the author knew the (math) braille for the element, they could not override `aria-label` by using the proposed aria-braillelabel because they wouldn't know which code to use.

As described in 2.1.6 Attributes Shared by all MathML Elements, MathML elements allow attributes `intent` and `arg` that allow the ***"intent"*** of the term to be specified. This annotation is not meant to provide a full mathematical definition of the term. It is primarily meant to help AT generate audio and/or braille renderings, see C. MathML Accessibility. Nevertheless, it may also be useful to guide translations to Content MathML, or computational systems.

The `intent` attribute encodes a simple functional syntax representing the intended speech. A formal grammar is given below, but a typical example would be `intent="power($base,$exponent)"` used in a context such as:

```
<msup intent="power($base,$exp)">
  <mi arg="base">x</mi>
  <mi arg="exp">n</mi>
</msup>
```

$$x^n$$

The `intent` value of `power($base,$exp)` makes it clear that the author intends that this expression denotes exponentiation as opposed to one of many other meanings of superscripts. Since power will be a concept supported by the AT, it may choose different ways of speaking depending on context, arguments or other details. For example, the above expression might be spoken as "x to the power n", but if "2" were given instead of "n", it may say "x squared".

5.1 The Grammar for `intent`

The value of the `intent` attribute, should match the following grammar.

```
intent             := self-property-list | expression
self-property-list := property+ S
expression         := S ( term property* | application ) S
```

```
term               := concept-or-literal | number | reference
concept-or-literal := NCName
number             := '-'? \d+ ( '.' \d+ )?
reference          := '$' NCName
application        := expression '(' arguments? S ')'
arguments          := expression ( ',' expression )*
property           := S ':' NCName
S                  := [ \t\n\r]*
```

Here `NCName` is as defined in in [xml-names], and `digit` is a character in the range 0–9.

The parts consist of:

concept-or-literal
: Names should match the `NCName` production as used for no-namespace element name. A concept-or-literal are interpreted either as a concept or literal.

- A ***concept*** corresponds to some mathematical or application specific function or concept. For many concepts, the words used to speak a concept are very similar to the name used when referencing a concept.
- A ***literal*** is a name starting with "`_`" (U+00F5). These will never be included in an Intent Concept Dictionary. The reading of a literal is generated by replacing any `-`, `_`, `.` in the name by spaces and then reading the resulting phrase.

number
: An explicit number such as `2.5` denotes itself.

reference
: An argument reference such as `$name` refers to a descendant element that has an attribute `arg="name"`. Unlike `id` attributes, `arg` do not have to be unique within a document. When searching for a matching element the search should only consider descendants, while stopping early at any elements that have a set `intent` or `arg` attribute, without descending into them. Proper use of reference, instead of inserting equivalent literals, allows intent to be used while navigating the mathematical structure.

application
: An application denotes a function applied to arguments using a standard prefix notation. Optionally, between the head of the function and the list of arguments there may be a property list as described below to influence the style of text reading generated, or to provide other information to any consumer of the intent.

property
: A property annotates the intent with an additional property which may be used by the system to adjust the generated speech or Braille in system specifc ways. The property may be directly related to the speech form, such as `:infix` or indirectly affect the style of speech with properties such as `:unit` or `:chemistry`

  The list of properties supported by any system is open but should include the core properties as described below.

self-property-list
: At the top level, an intent may consist of just a non-empty list of properties. These apply to the current element as described in 5.4 Using Intent Concepts and Properties.

expression
: A simple functional syntax using the terms described above.

5.2 Intent Concept Dictionaries

Every AT system that supports `intent` contains, at least implicitly, a list of the concepts that it recognizes. The details of matching and using concept names is given in 5.4 Using Intent Concepts and Properties. Such an AT *SHOULD* recognize the concepts in the Core list discussed below; It *MAY* also include concepts in the Open list discussed below, as well as any of its own.

An ***Intent Concept Dictionary*** is an abstract mapping of concept names to speech, text or braille for that concept; it is somewhat analogous to the B. Operator Dictionary used by MathML renderers in that it provides a set of defaults renderers should be aware of. The `property` also has some analogies to the operator dictionary's use of `form` because a match makes use of fixity properties (`prefix`, `infix`, etc.).

Intent Concept names are maintained in two lists, each maintained in the w3c/mathml-docs GitHub repository. Note that while these concept dictionaries are published as HTML tables (based on yaml data), there is no requirement on how a system implements the mapping from concepts to speech hints. Rather than a fixed list or hash table, it might use XPath matching, regular expressions, appropriately trained generative AI or any other suitable mechansim. The only requirement is that it should accept the cases listed in the Core concept dictionary and produce acceptable speech hints for those cases.

- ***Core***: This is a list of core concept names, initially drawn from concepts used in K14 STEM education. The entries include common concepts such as “divide”, “power”, and “greater-than”. The list is curated by the Math Working Group based on experience with different AT implementations and following the guidelines set out in [Concept-Lists].
- ***Open***: This is an open list of concepts to which contributions are invited. AT reading MathML attributed with a name in this list *MAY* use the speech hints provided by the intent definition but a system may also fall back on reading the identifier name as given. Although authors are encouraged to use a name in this list that matches their intent if one exists, any string that is an `NCName` is allowed.

Future versions of the “core” concept list may incorporate names from the “open” list if usage indicates that is appropriate.

5.3 Intent Properties

Intent properties act as modifiers of the speech or Braille that otherwise would have been generated by the `intent` attribute. Most of these properties only have a defined effect in specific contexts, such as on the head of an application or applying to an `<mtable>`. The use of these properties in other contexts is not an error, but as with any properties, is by default ignored but may have a system-specific effect.

As with Concepts, The Working group maintains two lists of property values.

- ***Core properties***: This is a list core of properties maintained by the Math Working Group
- ***Open properties***: This is an open list of properties with contributions welcome from the community. Implementors of MathML systems that implement additional properties are encouraged to make a pull request to add them to the list of Open Properties.

The definitive list of Core Properties is maintained at Github. Here, we describe the major classes of property affecting speech generation below.

`:prefix`, `:infix`, `:postfix`, `:function`, `:silent`
: These properties in a function application request that the reading of the name may be suppressed, or the word ordering

may be affected. Note that the properties `:prefix`, `:infix` and `:postfix` refer to the spoken word order of the name and arguments, and *not* (necessarily) the order used in the displayed mathematical notation.

- In the case of a supported concept name, the property *MAY* be used in choosing the alternatives supported by the AT. For example `union` is in the Core list with speech patterns "$1 union $2" and "union of $1 and $2". An intent `union :prefix ($a,$b)` would indicate that the latter style is preferred.
- For literal or unsupported concept names, the text generated from the function head *SHOULD* be read as specified in the property.
  - `f :prefix ($x)` : “f x”
  - `f :infix ($x,y)` : “x f y”
  - `f :postix ($x)` : “x f”
  - `f :function ($x, $y)`: “f of x and y”
  - `f :silent ($x,$y)` : “x y”

  The specific words used above are only examples; AT is free to choose other appropriate audio renderings. For example, `f:function($x, $y)` could also be spoken as “f of x comma y”. If none of these properties is used, the `function` property should be assumed unless the literal is silent (for example _) in which case the `:silent` property should be assumed. See the examples in 5.6 A Warning about literal and property.

`:literal`
This property requests that the AT should not infer any semantics and just speak the elements with a literal interpretation, including leaf content (eg “|” might be spoken as “vertical bar”).

`:matrix`, `:system-of-equations`, `:lines`, `:continued-equation`
These properties may be used on an `mtable` or on a reference to an `mtable`. They affect the way the parts of an alignment are announced.

The exact wordings used are system specfic

- `:matrix` should be read in style suitable for matricies, with typically column numbers being announced.
- `:system-of-equations` should be read in style suitable for displayed equations (and inequalities), with typically column numbers not being announced. Each table row would normally be announced as an "equation" but a `continued-equation` property on an `mtr` indicates that the row continues an equation wrapped from the row above.

5.4 Using Intent Concepts and Properties

When the `intent` attribute corresponding to a specific node contains a concept component, the AT's Intent Concept Dictionary should be consulted. The concept name should be normalized (“_” (U+00F5) and “`.`” (U+002E) to “–” (U+002D)), and compared using ASCII case-insensitive match. If arguments were given explicitly in the `intent` then their number gives the arity, and the fixity is determined from an explicit property or may default from the concept dictionary. Otherwise, arity is assumed to be 0.

A concept is considered a ***supported concept*** (by the AT) when the normalized name, the fixity property, and the arity all match an entry in the AT's concept dictionary. This does not exclude implementations which support additional concepts, as well as concepts with many arities, fixities or aliases, as long as they are mapped appropriately. The speech hint in the

matching entry can be used as a guide for the generation of specific audio, replacement text or braille renderings, as appropriate. It can also help clarify argument order. However, because common notations have many specialized ways of being spoken, the AT is NOT constrained to use the hint as given. For example, AT may vocalize a fraction marked up with `<mfrac>` as “three quarters” or “three over x” or may vocalize an inline fraction marked up as `<mo>/</mo>` as “three divided by x”. The choice may depend on the contents and carrier element associated with an `intent="divide($num, $denom)"`. Note that properties other than those specifying fixity may also indicate different rendering choices.

Otherwise, if the concept name, fixity and arity do not match that is considered to be an ***unsupported concept*** (by the AT) and will be treated the same as a literal; that is, the name is spoken as-is after normalizing each of `-`, `_` and `.` to an inter-word space. Even for an unsupported concept, if a fixity property and arguments were given, the speech for the arguments should be composed in a manner consistent with the given fixity property, if possible.

Note that future updates of an AT may add or remove concepts in its Intent Concept Dictionary. Hence which concepts are supported may change with each update.

In cases where the intent contains neither an explicit nor inferrable concept the AT should generally read out the MathML in a literal or structural fashion, as with the `:literal` property. However, any given properties should be respected if possible, and may be useful to indicate the kind of mathematical object, rather than giving an explicit concept name to be spoken. This can be a useful technique, especially for large constructs such as tables as it allows the children to be inferred without needing to be explicitly referenced in the `intent` as would be the case with an *applicaton*. For example, `<mtable intent=":array">...` might read the table as an array of values, whereas `<mtable intent=":system-of-equations">...` might read the table in a style more appropriate for a list of equations. In both cases the navigation of the underlying table structure can be supplied by the AT system, as it would for an unannotated table.

In general, depending upon the reader, AT may add words or sounds to make the speech clearer to the listener. For example, for someone who can not see the a fraction, AT might say “fraction x over three end fraction” so the listener knows exactly what is part of the fraction. For someone who can see the content, these extra words might be a distraction. AT should always produce speech that is appropriate to the community they serve.

5.5 Intent Error Handling

An intent processor may report errors in intent expressions in any appropriate way, including returning a message as the generated text, or throwing an exception (error) in whatever form the implementation supports. However in web platform contexts it is often not appropriate to report errors to the reader who has no access to correct the source, so intent procesors should offer a mode which recovers from errors as described below.

5.5.1 Intent Error Recovery

1. If an `intent` attribute does not match the grammar 5.1 The Grammar for `intent`, then the processor should act as if the attribute were not present. Typically this will result in a suitable fallback text being generated from the MathML element and its descendants. Note that just the erroneous attribute is ignored, other `intent` attributes in the MathML expression should be used.
2. If a `reference` such as `$x` does not correspond to an `arg` attribute with value `x` on a descendant element, the processor should act as if the reference were replaced by the literal `_dollar_x`.

5.6 A Warning about literal and property

The literal and property features extend the coverage of mathematical concepts beyond the predefined dictionaries and allow expression of speech preferences. For example, when `$x` and `$y` reference `<mi arg="x">x</mi>` and `<mi arg="y">y</mi>` respectively, then

- `list :silent ($x,$y)` would be read as “x y”
- `semi-factorial :postfix($x)` would be read as “x semi factorial”

These features also allow taking almost complete control of the generated speech. For example, compare:

- `free-algebra ($r, $x)`
  would be read as “free algebra of r and x”
- `free-algebra-construct:silent (_free, $r, _algebra, _on, $x)`
  would be read as “free r algebra on x”
- `_(free, _($r,algebra), on, $x)`
  would be read as “free r algebra; on x”

However, since the literals are not in dictionaries, the meaning behind the expressions become more opaque, and thus excessive use of these features will tend to limit the AT's ability to adapt to the needs of the user, as well as limit translation and locale-specific speech. Thus, the last two examples would be discouraged.

Conversely, when specific speech not corresponding to a meaningful concept is nevertheless required, it will be better to use a literal name (prefixed with “_”) rather than an unsupported concept. This avoids unexpected collisions with future updates to the concept dictionaries. Thus, the last example is particularly discouraged.

5.7 Intent Examples

A primary use for `intent` is to disambiguate cases where the same syntax is used for different meanings, and typically has different readings.

Superscript, `msup`, may represent a power, a transpose, a derivative or an embellished symbol. These cases would be distinguished as follows, showing possible readings *with* and *without* `intent`

```
<msup intent="power($base,$exp)">
  <mi arg="base">x</mi>
  <mi arg="exp">n</mi>
</msup>
```

> x to the n-th power
> x superscript n end superscript

$$x^n$$

An alternative default rendering without intent would be to assume that `msup` is always a power, so the second rendering above might also be “x to the n-th power”. In that case the second renderings below will (incorrectly) speak the examples using “raised to the ... power”.

```
<msup intent="$op($a)">
  <mi arg="a">A</mi>
  <mi arg="op" intent="transpose">T</mi>
</msup>
```

> transpose of A
> A superscript T end superscript

$$A^T$$

However, with a property, this example might be read differently.

```
<msup intent="$op :postfix ($a)">
  <mi arg="a">A</mi>
  <mi arg="op" intent="transpose">T</mi>
</msup>
```

> A transpose

$$A^T$$

```
<msup intent="derivative($a)">
  <mi arg="a">f</mi>
  <mi>′</mi>
</msup>
```

> derivative of f
> f superscript prime end superscript

$$f'$$

```
<msup intent="x-prime">
  <mi>x</mi>
  <mo>′</mo>
</msup>
```

> x prime
> x superscript prime end superscript

$$x'$$

Custom accessible descriptions, such as author-preferred variable or operator names, can also be annotated compositionally, via the underscore function.

The above notation could instead intend the custom name "x-new", which we can mark with a single literal `intent="_x-new"`, or as a compound narration of two arguments:

```
<msup intent="_($base,$script)">
  <mi arg="base">x</mi>
  <mo arg="script" intent="_new">′</mo>
</msup>
```

> x new
> x superscript prime end superscript

$$x'$$

Using the underscore function may also add clarity when the fragments of a compound name are explicitly localized. A cyrillic (Bulgarian) example:

```
<msup intent="_($base,$script)">
  <mi arg="base" intent="_хикс">x</mi>
  <mo arg="script" intent="_прим">′</mo>
</msup>
```

> хикс прим
> x superscript prime end superscript

$$x'$$

Alternatively, the narration of individual fragments could be fully delegated to AT, while still specifying their grouping:

```
<msup intent="_($base,$script)">
  <mi arg="base">x</mi>
  <mo arg="script">′</mo>
</msup>
```

> x prime
> x superscript prime end superscript

$$x'$$

An overbar may represent complex conjugation, or mean (average), again with possible readings with and without `intent`:

```
<mover intent="conjugate($v)">
  <mi arg="v">z</mi>
  <mo>&#xaf;</mo>
</mover>
<mtext> <!--nbsp-->is not <!--nbsp--></mtext>
<mover intent="mean($var)">
  <mi arg="var">X</mi>
  <mo>&#xaf;</mo>
</mover>
```

> conjugate of z is not mean of X
> z with bar above is not X with bar above

$$\overline{z} \text{ is not } \overline{X}$$

The intent mechanism is extensible through the use of unsupported concept names. For example, assuming that the Bell Number is not present in any of the dictionaries, the following example

```
<msub intent="bell-number($index)">
  <mi>B</mi>
  <mn arg="index">2</mn>
</msub>
```

will still produce the expected reading:

> bell number of 2

$$B_2$$

5.7.1 CSS and Style

CSS customization of MathML is generally not made available to AT and is ignored in accessible readouts. In cases where authors have meaningful stylistic emphases, or stylized constructs with custom names, using an intent attribute is appropriate. For example, color-coding of subexpressions is often helpful in teaching materials:

```
<mn>1</mn><mo>+</mo>
<mrow style="padding:0.1em;background-color:lightyellow;"
      intent="highlighted-step($step)">
  <mfrac arg="step"><mn>6</mn><mn>2</mn></mfrac>
</mrow>
<mi>x</mi>
<mo>=</mo>
<mn>1</mn><mo>+</mo>
<mn style="padding:0.1em;background-color:lightgreen;"
    intent="highlighted-result(3)">3</mn>
<mi>x</mi>
```

> one plus highlighted step of six over two end highlighted step x equals one plus highlighted result of three end highlighted result x

$$1 + \frac{6}{2}\,x = 1 + 3x$$

5.7.2 Tables

The `<mtable>` element is used in many ways, for denoting matrices, systems of equations, steps in a proof derivation, etc. In addition to these uses it may be used to implement forced line breaking and alignment, especially for systems that do not implement 3.1.7 Linebreaking of Expressions, or for conversions from (La)TeX where alignment constructs are used in similar ways.

Whenever a kind of tabular construct has an associated property, it is usually better to use only the property and allow AT to infer how to speak navigate the expression. By use of properties in this way the author can give hints to the speech generation and generate speech suitable for a list of aligned equations rather than say a matrix.

When core properties are insufficient to represent a tabular layout, the use of intent concept names and, if appropriate, also properties from the open list of properties should be used to convey the desired speech and navigation of the tabular layout. Because of the likely complexity of these layouts, testing with AT should be done to verify that users hear the expression as the author intended.

Matrices

```
<mrow intent='$m'>
  <mo>(</mo>
  <mtable arg='m' intent=':matrix'>
    <mtr>
      <mtd><mn>1</mn></mtd>
      <mtd><mn>0</mn></mtd>
    </mtr>
    <mtr>
      <mtd><mn>0</mn></mtd>
      <mtd><mn>1</mn></mtd>
    </mtr>
  </mtable>
  <mo>)</mo>
</mrow>
```

> The 2 by 2 matrix;
> column 1; 1;
> column 2; 0;
> column 1; 0;
> column 2; 1;
> end matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Aligned equations

```
<mtable intent=':equations'>
  <mtr>
    <mtd columnalign="right">
      <mn>2</mn>
      <mo>&#x2062;<!--InvisibleTimes--></mo>
      <mi>x</mi>
    </mtd>
    <mtd columnalign="center">
      <mo>=</mo>
    </mtd>
    <mtd columnalign="left">
      <mn>1</mn>
    </mtd>
  </mtr>
  <mtr>
    <mtd columnalign="right">
      <mi>y</mi>
    </mtd>
    <mtd columnalign="center">
      <mo>></mo>
    </mtd>
    <mtd columnalign="left">
      <mi>x</mi>
      <mo>-</mo>
      <mn>3</mn>
    </mtd>
  </mtr>
</mtable>
```

2 equations,
equation 1; 2 x, is equal to, 1;
equation 2; y, is greater than, x minus 3;

$$\begin{aligned} 2x &= 1 \\ y &> x - 3 \end{aligned}$$

Aligned Equations with wrapped expressions

```
<mtable intent=':equations'>
  <mtr>
    <mtd columnalign="right">
      <mi>a</mi>
    </mtd>
    <mtd columnalign="center">
      <mo>=</mo>
    </mtd>
    <mtd columnalign="left">
      <mi>b</mi>
      <mo>+</mo>
      <mi>c</mi>
      <mo>-</mo>
      <mi>d</mi>
    </mtd>
  </mtr>
  <mtr intent=':continued-equation'>
    <mtd columnalign="right"></mtd>
    <mtd columnalign="center"></mtd>
    <mtd columnalign="left">
      <mo form="infix">+</mo>
      <mi>e</mi>
      <mo>-</mo>
      <mi>f</mi>
    </mtd>
  </mtr>
</mtable>
```

> 1 equation; a, is equal to, b plus c minus d; plus e minus f;

$$\begin{aligned} a &= b + c - d \\ &\quad + e - f \end{aligned}$$

6. Annotating MathML: semantics

In addition to the `intent` attribute described above, MathML provides a more general framework for annotation. A MathML expression may be decorated with a sequence of pairs made up of a symbol that indicates the kind of annotation, known as the *annotation key*, and associated data, known as the *annotation value*.

The `semantics`, `annotation`, and `annotation-xml` elements are used together to represent annotations in MathML. The `semantics` element provides the container for an expression and its annotations. The `annotation` element is the container for text annotations, and the `annotation-xml` element is used for structured annotations. The `semantics` element contains the expression being annotated as its first child, followed by a sequence of zero or more `annotation` and/or `annotation-xml` elements.

The `semantics` element is considered to be both a presentation element and a content element, and may be used in either

context. All MathML processors should process the `semantics` element, even if they only process one of these two subsets of MathML, or [MathML-Core].

6.1 Annotation keys

An *annotation key* specifies the relationship between an expression and an annotation. Many kinds of relationships are possible. Examples include alternate representations, specification or clarification of semantics, type information, rendering hints, and private data intended for specific processors. The annotation key is the primary means by which a processor determines whether or not to process an annotation.

The logical relationship between an expression and an annotation can have a significant impact on the proper processing of the expression. For example, a particular annotation form, called *semantic attributions*, cannot be ignored without altering the meaning of the annotated expression, at least in some processing contexts. On the other hand, alternate representations do not alter the meaning of an expression, but may alter the presentation of the expression as they are frequently used to provide rendering hints. Still other annotations carry private data or metadata that are useful in a specific context, but do not alter either the semantics or the presentation of the expression.

Annotation keys may be defined as a symbol in a Content Dictionary, and are specified using the `cd` and `name` attributes on the `annotation` and `annotation-xml` elements. Alternatively, an annotation key may also be referenced using the `definitionURL` attribute as an alternative to the `cd` and `name` attributes.

MathML provides two predefined annotation keys for the most common kinds of annotations: alternate-representation and contentequiv defined in the mathmlkeys content dictionary. The alternate-representation annotation key specifies that the annotation value provides an alternate representation for an expression in some other markup language, and the contentequiv annotation key specifies that the annotation value provides a semantically equivalent alternative for the annotated expression.

The default annotation key is alternate-representation when no annotation key is explicitly specified on an `annotation` or `annotation-xml` element.

Typically, annotation keys specify only the logical nature of the relationship between an expression and an annotation. The data format for an annotation is indicated with the `encoding` attribute. In MathML 2, the `encoding` attribute was the primary information that a processor could use to determine whether or not it could understand an annotation. For backward compatibility, processors are encouraged to examine both the annotation key and `encoding` attribute. In particular, MathML 2 specified the predefined encoding values `MathML`, `MathML-Content`, and `MathML-Presentation`. The `MathML` encoding value is used to indicate an `annotation-xml` element contains a MathML expression. The use of the other values is more specific, as discussed in following sections.

6.2 Alternate representations

Alternate representation annotations are most often used to provide renderings for an expression, or to provide an equivalent representation in another markup language. In general, alternate representation annotations do not alter the meaning of the annotated expression, but may alter its presentation.

A particularly important case is the use of a presentation MathML expression to indicate a preferred rendering for a content

MathML expression. This case may be represented by labeling the annotation with the `application/mathml-presentation+xml` value for the `encoding` attribute. For backward compatibility with MathML 2.0, this case can also be represented with the equivalent `MathML-Presentation` value for the `encoding` attribute. Note that when a presentation MathML annotation is present in a `semantics` element, it may be used as the default rendering of the `semantics` element, instead of the default rendering of the first child.

In the example below, the `semantics` element binds together various alternate representations for a content MathML expression. The presentation MathML annotation may be used as the default rendering, while the other annotations give representations in other markup languages. Since no attribution keys are explicitly specified, the default annotation key alternate-representation applies to each of the annotations.

```
<semantics>
  <apply>
    <plus/>
    <apply><sin/><ci>x</ci></apply>
    <cn>5</cn>
  </apply>
  <annotation-xml encoding="MathML-Presentation">
    <mrow>
      <mrow>
        <mi>sin</mi>
        <mo>&#x2061;<!--ApplyFunction--></mo>
        <mrow><mo>(</mo><mi>x</mi><mo>)</mo></mrow>
      </mrow>
      <mo>+</mo>
      <mn>5</mn>
    </mrow>
  </annotation-xml>
  <annotation encoding="application/x-maple">sin(x) + 5</annotation>
  <annotation encoding="application/vnd.wolfram.mathematica">Sin[x] + 5</annotation>
  <annotation encoding="application/x-tex">\sin x + 5</annotation>
  <annotation-xml encoding="application/openmath+xml">
    <OMA xmlns="http://www.openmath.org/OpenMath">
      <OMA>
        <OMS cd="arith1" name="plus"/>
        <OMA><OMS cd="transc1" name="sin"/><OMV name="x"/></OMA>
        <OMI>5</OMI>
      </OMA>
    </OMA>
  </annotation-xml>
</semantics>
```

Note that this example makes use of the namespace extensibility that is only available in the XML syntax of MathML. If this example is included in an HTML document then it would be considered *invalid* and the OpenMath elements would be parsed as elements in the MathML namespace. See 6.7.3 Using `annotation-xml` in HTML documents for details.

6.3 Content equivalents

Content equivalent annotations provide additional computational information about an expression. Annotations with the contentequiv key cannot be ignored without potentially changing the behavior of an expression.

An important case arises when a content MathML annotation is used to disambiguate the meaning of a presentation MathML expression. This case may be represented by labeling the annotation with the `application/mathml-content+xml` value for the `encoding` attribute. In MathML 2, this type of annotation was represented with the equivalent `MathML-Content` value for the `encoding` attribute, so processors are urged to support this usage for backward compatibility. The contentequiv annotation key should be used to make an explicit assertion that the annotation provides a definitive content markup equivalent for an expression.

In the example below, an ambiguous presentation MathML expression is annotated with a `MathML-Content` annotation clarifying its precise meaning.

```
<semantics>
  <mrow>
    <mrow>
      <mi>a</mi>
      <mrow>
        <mo>(</mo>
        <mrow><mi>x</mi><mo>+</mo><mn>5</mn></mrow>
        <mo>)</mo>
      </mrow>
    </mrow>
  </mrow>
  <annotation-xml cd="mathmlkeys" name="contentequiv" encoding="MathML-Content">
    <apply>
      <ci>a</ci>
      <apply><plus/><ci>x</ci><cn>5</cn></apply>
    </apply>
  </annotation-xml>
</semantics>
```

6.4 Annotation references

In the usual case, each annotation element includes either character data content (in the case of `annotation`) or XML markup data (in the case of `annotation-xml`) that represents the *annotation value*. There is no restriction on the type of annotation that may appear within a `semantics` element. For example, an annotation could provide a TeX encoding, a linear input form for a computer algebra system, a rendered image, or detailed mathematical type information.

In some cases the alternative children of a `semantics` element are not an essential part of the behavior of the annotated expression, but may be useful to specialized processors. To enable the availability of several annotation formats in a more efficient manner, a `semantics` element may contain empty `annotation` and `annotation-xml` elements that provide `encoding` and `src` attributes to specify an external location for the annotation value associated with the annotation. This type of annotation is known as an *annotation reference*.

```
<semantics>
  <mfrac><mi>a</mi><mrow><mi>a</mi><mo>+</mo><mi>b</mi></mrow></mfrac>
  <annotation encoding="image/png" src="333/formula56.png"/>
  <annotation encoding="application/x-maple" src="333/formula56.ms"/>
</semantics>
```

Processing agents that anticipate that consumers of exported markup may not be able to retrieve the external entity referenced by such annotations should request the content of the external entity at the indicated location and replace the annotation with its expanded form.

An annotation reference follows the same rules as for other annotations to determine the annotation key that specifies the relationship between the annotated object and the annotation value.

6.5 The <semantics> element

6.5.1 Description

The `semantics` element is the container element that associates annotations with a MathML expression. The `semantics` element has as its first child the expression to be annotated. Any MathML expression may appear as the first child of the `semantics` element. Subsequent `annotation` and `annotation-xml` children enclose the annotations. An annotation represented in XML is enclosed in an `annotation-xml` element. An annotation represented in character data is enclosed in an `annotation` element.

As noted above, the `semantics` element is considered to be both a presentation element and a content element, since it can act as either, depending on its content. Consequently, all MathML processors should process the `semantics` element, even if they process only presentation markup or only content markup.

The default rendering of a `semantics` element is the default rendering of its first child. A renderer may use the information contained in the annotations to customize its rendering of the annotated element.

```
<semantics>
  <mrow>
    <mrow>
      <mi>sin</mi>
      <mo>&#x2061;<!--ApplyFunction--></mo>
      <mrow><mo>(</mo><mi>x</mi><mo>)</mo></mrow>
    </mrow>
    <mo>+</mo>
    <mn>5</mn>
  </mrow>
  <annotation-xml cd="mathmlkeys" name="contentequiv" encoding="MathML-Content">
    <apply>
      <plus/>
      <apply><sin/><ci>x</ci></apply>
      <cn>5</cn>
    </apply>
  </annotation-xml>
  <annotation encoding="application/x-tex">\sin x + 5</annotation>
</semantics>
```

$$\sin(x) + 5$$

6.6 The <annotation> element

6.6.1 Description

The annotation element is the container element for a semantic annotation whose representation is parsed character data in a non-XML format. The annotation element should contain the character data for the annotation, and should not contain XML markup elements. If the annotation contains one of the XML reserved characters &, < then these characters must be encoded using an entity reference or (in the XML syntax) an XML CDATA section.

6.6.2 Attributes

Name	values	default
definitionURL	*URI*	*none*
	The location of the annotation key symbol	
encoding	*string*	*none*
	The encoding of the semantic information in the annotation	
cd	*string*	mathmlkeys
	The content dictionary that contains the annotation key symbol	
name	*string*	alternate-representation
	The name of the annotation key symbol	
src	*URI*	*none*
	The location of an external source for semantic information	

Taken together, the `cd` and `name` attributes specify the annotation key symbol, which identifies the relationship between the annotated element and the annotation, as described in 6.5 The `<semantics>` element. The `definitionURL` attribute provides an alternate way to reference the annotation key symbol as a single attribute. If none of these attributes are present, the annotation key symbol is the symbol alternate-representation from the mathmlkeys content dictionary.

The `encoding` attribute describes the content type of the annotation. The value of the `encoding` attribute may contain a media type that identifies the data format for the encoding data. For data formats that do not have an associated media type, implementors may choose a self-describing character string to identify their content type.

The `src` attribute provides a mechanism to attach external entities as annotations on MathML expressions.

```
<annotation encoding="image/png" src="333/formula56.png"/>
```

The `annotation` element is a semantic mapping element that may only be used as a child of the `semantics` element. While there is no default rendering for the `annotation` element, a renderer may use the information contained in an annotation to customize its rendering of the annotated element.

6.7 The `<annotation-xml>` element

6.7.1 Description

The `annotation-xml` element is the container element for a semantic annotation whose representation is structured markup. The `annotation-xml` element should contain the markup elements, attributes, and character data for the

annotation.

6.7.2 Attributes

Name	values	default
definitionURL	*URI*	*none*
	The location of the annotation key symbol	
encoding	*string*	*none*
	The encoding of the semantic information in the annotation	
cd	*string*	mathmlkeys
	The content dictionary that contains the annotation key symbol	
name	*string*	alternate-representation
	The name of the annotation key symbol	
src	*URI*	*none*
	The location of an external source for semantic information	

Taken together, the `cd` and `name` attributes specify the annotation key symbol, which identifies the relationship between the annotated element and the annotation, as described in 6.5 The `<semantics>` element. The `definitionURL` attribute provides an alternate way to reference the annotation key symbol as a single attribute. If none of these attributes are present, the annotation key symbol is the symbol alternate-representation from the mathmlkeys content dictionary.

The `encoding` attribute describes the content type of the annotation. The value of the `encoding` attribute may contain a media type that identifies the data format for the encoding data. For data formats that do not have an associated media type, implementors may choose a self-describing character string to identify their content type. In particular, as described above and in 7.2.4 Names of MathML Encodings, MathML specifies `MathML`, `MathML-Presentation`, and `MathML-Content` as predefined values for the `encoding` attribute. Finally, the `src` attribute provides a mechanism to attach external XML entities as annotations on MathML expressions.

```
<annotation-xml cd="mathmlkeys" name="contentequiv" encoding="MathML-Content">
  <apply>
    <plus/>
    <apply><sin/><ci>x</ci></apply>
    <cn>5</cn>
  </apply>
</annotation-xml>

<annotation-xml encoding="application/openmath+xml">
  <OMA xmlns="http://www.openmath.org/OpenMath">
    <OMS cd="arith1" name="plus"/>
    <OMA><OMS cd="transc1" name="sin"/><OMV name="x"/></OMA>
    <OMI>5</OMI>
  </OMA>
</annotation-xml>
```

When the MathML is being parsed as XML and the annotation value is represented in an XML dialect other than MathML, the namespace for the XML markup for the annotation should be identified by means of namespace attributes and/or namespace prefixes on the annotation value. For instance:

```
<annotation-xml encoding="application/xhtml+xml">
  <html xmlns="http://www.w3.org/1999/xhtml">
    <head><title>E</title></head>
    <body>
      <p>The base of the natural logarithms, approximately 2.71828.</p>
    </body>
  </html>
</annotation-xml>
```

The `annotation-xml` element is a semantic mapping element that may only be used as a child of the `semantics` element. While there is no default rendering for the `annotation-xml` element, a renderer may use the information contained in an annotation to customize its rendering of the annotated element.

6.7.3 Using `annotation-xml` in HTML documents

Note that the Namespace extensibility used in the above examples may not be available if the MathML is not being treated as an XML document. In particular HTML parsers treat `xmlns` attributes as ordinary attributes, so the OpenMath example would be classified as invalid by an HTML validator. The OpenMath elements would still be parsed as children of the `annotation-xml` element, however they would be placed in the *MathML* namespace. The above examples are not rendered in the HTML version of this specification, to ensure that that document is a valid HTML5 document.

The details of the HTML parser handling of `annotation-xml` is specified in [HTML] and summarized in 7.4.3 Mixing MathML and HTML, however the main differences from the behavior of an XML parser that affect MathML annotations are that the HTML parser does not treat `xmlns` attributes, nor `:` in element names as special and has built-in rules determining whether the three “known” namespaces, HTML, SVG or MathML are used.

- If the `annotation-xml` has an `encoding` attribute that is (ignoring case differences) `text/html` or `annotation/`

`xhtml+xml` then the content is parsed as HTML and placed (initially) in the HTML namespace.

- Otherwise it is parsed as *foreign content* and parsed in a more XML-like manner (like MathML itself in HTML) in which `/>` signifies an empty element. Content will be placed in the MathML namespace.

  If any recognised HTML element appears in this foreign content annotation the HTML parser will effectively terminate the math expression, closing all open elements until the `math` element is closed, and then process the nested HTML as if it were not inside the math context. Any following MathML elements will then not render correctly as they are not in a math context, or in the MathML namespace.

These issues mean that the following example is valid whether parsed by an XML or HTML parser:

```
<math>
  <semantics>
    <mi>a</mi>
    <annotation-xml encoding="text/html">
      <span>xxx</span>
    </annotation-xml>
  </semantics>
  <mo>+</mo>
  <mi>b</mi>
</math>
```

However if the `encoding` attribute is omitted then the expression is only valid if parsed as XML:

```
<math>
  <semantics>
    <mi>a</mi>
    <annotation-xml>
      <span>xxx</span>
    </annotation-xml>
  </semantics>
  <mo>+</mo>
  <mi>b</mi>
</math>
```

If the above is parsed by an HTML parser it produces a result equivalent to the following invalid input, where the `span` element has caused all MathML elements to be prematurely closed. The remaining MathML elements following the `span` are no longer contained within `<math>` so will be parsed as unknown HTML elements and render incorrectly.

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <semantics>
    <mi>a</mi>
    <annotation-xml>
    </annotation-xml>
  </semantics>
</math>
<span xmlns="http://www.w3.org/1999/xhtml">xxx</span>
<mo xmlns="http://www.w3.org/1999/xhtml">+</mo>
<mi xmlns="http://www.w3.org/1999/xhtml">b</mi>
```

Note here that the HTML `span` element has caused all open MathML elements to be prematurely closed, resulting in the following MathML elements being treated as unknown HTML elements as they are no longer descendants of `math`. See 7.4.3 Mixing MathML and HTML for more details of the parsing of MathML in HTML.

Any use of elements in other vocabularies (such as the OpenMath examples above) is considered *invalid* in HTML. If validity is not a strict requirement it is possible to use such elements but they will be parsed as elements on the *MathML* namespace. Documents *SHOULD NOT* use namespace prefixes and element names containing colon (`:`) as the element nodes produced by the HTML parser have local names containing a colon, which can not be constructed by a namespace aware XML parser. Rather than use such foreign annotations, when using an HTML parser it is better to encode the annotation using the existing vocabulary. For example as shown in 4. Content Markup OpenMath may be encoded faithfully as *Strict Content MathML*. Similarly RDF annotations could be encoded using RDFa in `text/html` annotation or (say) N3 notation in `annotation` rather than using RDF/XML encoding in an `annotation-xml` element.

6.8 Combining Presentation and Content Markup

Presentation markup encodes the *notational structure* of an expression. Content markup encodes the *functional structure* of an expression. In certain cases, a particular application of MathML may require a combination of both presentation and content markup. This section describes specific constraints that govern the use of presentation markup within content markup, and vice versa.

6.8.1 Presentation Markup in Content Markup

Presentation markup may be embedded within content markup so long as the resulting expression retains an unambiguous function application structure. Specifically, presentation markup may only appear in content markup in three ways:

1. within `ci` and `cn` token elements
2. within the `csymbol` element
3. within the `semantics` element

Any other presentation markup occurring within content markup is a MathML error. More detailed discussion of these three cases follows:

Presentation markup within token elements.

The token elements `ci` and `cn` are permitted to contain any sequence of MathML characters (defined in 8. Characters, Entities and Fonts) and/or presentation elements. Contiguous blocks of MathML characters in `ci` or `cn` elements are treated as if wrapped in `mi` or `mn` elements, as appropriate, and the resulting collection of presentation elements is rendered as if wrapped in an implicit `mrow` element.

Presentation markup within the `csymbol` element.
The same rendering rules that apply to the token elements `ci` and `cn` should be used for the `csymbol` element.

Presentation markup within the `semantics` element.
One of the main purposes of the `semantics` element is to provide a mechanism for incorporating arbitrary MathML expressions into content markup in a semantically meaningful way. In particular, any valid presentation expression can be embedded in a content expression by placing it as the first child of a `semantics` element. The meaning of this wrapped expression should be indicated by one or more annotation elements also contained in the `semantics` element.

6.8.2 Content Markup in Presentation Markup

Content markup may be embedded within presentation markup so long as the resulting expression has an unambiguous rendering. That is, it must be possible, in principle, to produce a presentation markup fragment for each content markup fragment that appears in the combined expression. The replacement of each content markup fragment by its corresponding presentation markup should produce a well-formed presentation markup expression. A presentation engine should then be able to process this presentation expression without reference to the content markup bits included in the original expression.

In general, this constraint means that each embedded content expression must be well-formed, as a content expression, and must be able to stand alone outside the context of any containing content markup element. As a result, the following content elements may not appear as an immediate child of a presentation element: `annotation`, `annotation-xml`, `bvar`, `condition`, `degree`, `logbase`, `lowlimit`, `uplimit`.

In addition, within presentation markup, content markup may not appear within presentation token elements.

6.9 Parallel Markup

Some applications are able to use *both* presentation and content information. *Parallel markup* is a way to combine two or more markup trees for the same mathematical expression. Parallel markup is achieved with the `semantics` element. Parallel markup for an expression may appear on its own, or as part of a larger content or presentation tree.

6.9.1 Top-level Parallel Markup

In many cases, the goal is to provide presentation markup and content markup for a mathematical expression as a whole. A single `semantics` element may be used to pair two markup trees, where one child element provides the presentation markup, and the other child element provides the content markup.

The following example encodes the Boolean arithmetic expression $(a + b)(c + d)$ in this way.

```
<semantics>
  <mrow>
    <mrow><mo>(</mo><mi>a</mi> <mo>+</mo> <mi>b</mi><mo>)</mo></mrow>
    <mo>&#x2062;<!--InvisibleTimes--></mo>
    <mrow><mo>(</mo><mi>c</mi> <mo>+</mo> <mi>d</mi><mo>)</mo></mrow>
  </mrow>
  <annotation-xml encoding="MathML-Content">
    <apply><and/>
      <apply><xor/><ci>a</ci> <ci>b</ci></apply>
      <apply><xor/><ci>c</ci> <ci>d</ci></apply>
    </apply>
  </annotation-xml>
</semantics>
```

$$(a + b)(c + d)$$

Note that the above markup annotates the presentation markup as the first child element, with the content markup as part of the `annotation-xml` element. An equivalent form could be given that annotates the content markup as the first child element, with the presentation markup as part of the `annotation-xml` element.

6.9.2 Parallel Markup via Cross-References

To accommodate applications that must process sub-expressions of large objects, MathML supports cross-references between the branches of a `semantics` element to identify corresponding sub-structures. These cross-references are established by the use of the `id` and `xref` attributes within a `semantics` element. This application of the `id` and `xref` attributes within a `semantics` element should be viewed as best practice to enable a recipient to select arbitrary sub-expressions in each alternative branch of a `semantics` element. The `id` and `xref` attributes may be placed on MathML elements of any type.

The following example demonstrates cross-references for the Boolean arithmetic expression $(a + b)(c + d)$.

```
<semantics>
  <mrow id="E">
    <mrow id="E.1">
      <mo id="E.1.1">(</mo>
      <mi id="E.1.2">a</mi>
      <mo id="E.1.3">+</mo>
      <mi id="E.1.4">b</mi>
      <mo id="E.1.5">)</mo>
    </mrow>
    <mo id="E.2">&#x2062;<!--InvisibleTimes--></mo>
    <mrow id="E.3">
      <mo id="E.3.1">(</mo>
      <mi id="E.3.2">c</mi>
      <mo id="E.3.3">+</mo>
      <mi id="E.3.4">d</mi>
      <mo id="E.3.5">)</mo>
    </mrow>
  </mrow>

  <annotation-xml encoding="MathML-Content">
    <apply xref="E">
      <and xref="E.2"/>
      <apply xref="E.1">
        <xor xref="E.1.3"/><ci xref="E.1.2">a</ci><ci xref="E.1.4">b</ci>
      </apply>
      <apply xref="E.3">
        <xor xref="E.3.3"/><ci xref="E.3.2">c</ci><ci xref="E.3.4">d</ci>
      </apply>
    </apply>
  </annotation-xml>
</semantics>
```

$$(a + b)(c + d)$$

An `id` attribute and associated `xref` attributes that appear within the same `semantics` element establish the cross-references between corresponding sub-expressions.

For parallel markup, all of the `id` attributes referenced by any `xref` attribute should be in the *same* branch of an enclosing `semantics` element. This constraint guarantees that the cross-references do not create unintentional cycles. This restriction does *not* exclude the use of `id` attributes within other branches of the enclosing `semantics` element. It does, however, exclude references to these other `id` attributes originating from the same `semantics` element.

There is no restriction on which branch of the `semantics` element may contain the destination `id` attributes. It is up to the application to determine which branch to use.

In general, there will not be a one-to-one correspondence between nodes in parallel branches. For example, a presentation tree may contain elements, such as parentheses, that have no correspondents in the content tree. It is therefore often useful to put the `id` attributes on the branch with the finest-grained node structure. Then all of the other branches will have `xref` attributes to some subset of the `id` attributes.

In absence of other criteria, the first branch of the `semantics` element is a sensible choice to contain the `id` attributes. Applications that add or remove annotations will then not have to re-assign these attributes as the annotations change.

In general, the use of `id` and `xref` attributes allows a full correspondence between sub-expressions to be given in text that is at most a constant factor larger than the original. The direction of the references should not be taken to imply that sub-expression selection is intended to be permitted only on one child of the `semantics` element. It is equally feasible to select a subtree in any branch and to recover the corresponding subtrees of the other branches.

Parallel markup with cross-references may be used in any of the semantic annotations within `annotation-xml`, for example cross referencing between a presentation MathML rendering and an OpenMath annotation.

As noted above, the use of namespaces other than MathML, SVG or HTML within `annotation-xml` is not considered valid in the HTML syntax. Use of colons and namespace-prefixed element names should be avoided as the HTML parser will generate nodes with *local* name `om:OMA` (for example), and such nodes can not be constructed by a namespace-aware XML parser.

7. Interactions with the Host Environment

7.1 Introduction

To be effective, MathML must work well with a wide variety of renderers, processors, translators and editors. This chapter raises some of the interface issues involved in generating and rendering MathML. Since MathML exists primarily to encode mathematics in Web documents, perhaps the most important interface issues relate to embedding MathML in [HTML], and [XHTML], and in any newer HTML when it appears.

There are two kinds of interface issues that arise in embedding MathML in other XML documents. First, MathML markup must be recognized as valid embedded XML content, and not as an error. This issue could be seen primarily as a question of managing namespaces in XML [Namespaces].

Second, tools for generating and processing MathML must be able to reliably communicate. MathML tools include editors, translators, computer algebra systems, and other scientific software. However, since MathML expressions tend to be lengthy, and prone to error when entered by hand, special emphasis must be made to ensure that MathML can easily be generated by user-friendly conversion and authoring tools, and that these tools work together in a dependable, platform-independent, and vendor-independent way.

This chapter applies to both content and presentation markup, and describes a particular processing model for the `semantics`, `annotation` and `annotation-xml` elements described in 6. Annotating MathML: semantics.

7.2 Invoking MathML Processors

7.2.1 Recognizing MathML in XML

Within an XML document supporting namespaces [XML], [Namespaces], the preferred method to recognize MathML markup is by the identification of the `math` element in the MathML namespace by the use of the MathML namespace URI `http://www.w3.org/1998/Math/MathML`.

The MathML namespace URI is the recommended method to embed MathML within [XHTML] documents. However, some user-agents may require supplementary information to be available to allow them to invoke specific extensions to process the MathML markup.

Markup-language specifications that wish to embed MathML may require special conditions to recognize MathML markup that are independent of this recommendation. The conditions should be similar to those expressed in this recommendation, and the local names of the MathML elements should remain the same as those defined in this recommendation.

7.2.2 Recognizing MathML in HTML

HTML does not allow arbitrary namespaces, but has built in knowledge of the MathML namespace. The `math` element and its descendants will be placed in the `http://www.w3.org/1998/Math/MathML` namespace by the HTML parser, and will appear to applications as if the input had been XHTML with the namespace declared as in the previous section. See 7.4.3 Mixing MathML and HTML for detailed rules of the HTML parser's handling of MathML.

7.2.3 Resource Types for MathML Documents

Although rendering MathML expressions often takes place in a Web browser, other MathML processing functions take place more naturally in other applications. Particularly common tasks include opening a MathML expression in an equation editor or computer algebra system. It is important therefore to specify the encoding names by which MathML fragments should be identified.

Outside of those environments where XML namespaces are recognized, media types [RFC2045], [RFC2046] should be used if possible to ensure the invocation of a MathML processor. For those environments where media types are not appropriate, such as clipboard formats on some platforms, the encoding names described in the next section should be used.

7.2.4 Names of MathML Encodings

MathML contains two distinct vocabularies: one for encoding visual presentation, defined in 3. Presentation Markup, and one for encoding computational structure, defined in 4. Content Markup. Some MathML applications may import and export only one of these two vocabularies, while others may produce and consume each in a different way, and still others may process both without any distinction between the two. The following encoding names may be used to distinguish between content and presentation MathML markup when needed.

- *MathML-Presentation*: The instance contains presentation MathML markup only.
  - Media Type: `application/mathml-presentation+xml`
  - Windows Clipboard Flavor: `MathML Presentation`

- Universal Type Identifier: `public.mathml.presentation`

- *MathML-Content*: The instance contains content MathML markup only.
  - Media Type: `application/mathml-content+xml`
  - Windows Clipboard Flavor: `MathML Content`
  - Universal Type Identifier: `public.mathml.content`
- *MathML* (generic): The instance may contain presentation MathML markup, content MathML markup, or a mixture of the two.
  - File name extension: `.mml`
  - Media Type: `application/mathml+xml`
  - Windows Clipboard Flavor: `MathML`
  - Universal Type Identifier: `public.mathml`

See [MathML-Media-Types] for more details about each of these encoding names.

MathML 2 specified the predefined encoding values `MathML`, `MathML-Content`, and `MathML-Presentation` for the `encoding` attribute on the `annotation-xml` element. These values may be used as an alternative to the media type for backward compatibility. See 6.2 Alternate representations and 6.3 Content equivalents for details. Moreover, MathML 1.0 suggested the media-type `text/mathml`, which has been superseded by [RFC7303].

7.3 Transferring MathML

MathML expressions are often exchanged between applications using the familiar copy-and-paste or drag-and-drop paradigms and are often stored in files or exchanged over the HTTP protocol. This section provides recommended ways to process MathML during these transfers.

The transfers of MathML fragments described in this section occur between the contexts of two applications by making the MathML data available in several flavors, often called *media types*, *clipboard formats*, or *data flavors*. These flavors are typically ordered by preference by the producing application, and are typically examined in preference order by the consuming application. The copy-and-paste paradigm allows an application to *place* content in a central *clipboard*, with one data stream per *clipboard format*; a consuming application negotiates by choosing to read the data of the format it prefers. The drag-and-drop paradigm allows an application to *offer* content by declaring the available formats; a potential recipient accepts or rejects a drop based on the list of available formats, and the drop action allows the receiving application to request the delivery of the data in one of the indicated formats. An HTTP GET transfer, as in [rfc9110], allows a client to submit a list of acceptable media types; the server then delivers the data using one of the indicated media types. An HTTP POST transfer, as in [rfc9110], allows a client to submit data labelled with a media type that is acceptable to the server application.

Current desktop platforms offer copy-and-paste and drag-and-drop transfers using similar architectures, but with varying naming schemes depending on the platform. HTTP transfers are all based on media types. This section specifies what transfer types applications should provide, how they should be named, and how they should handle the special `semantics`, `annotation`, and `annotation-xml` elements.

To summarize the three negotiation mechanisms, the following paragraphs will describe transfer *flavors*, each with a *name* (a character string) and *content* (a stream of binary data), which are *offered*, *accepted*, and/or *exported*.

7.3.1 Basic Transfer Flavor Names and Contents

The names listed in 7.2.4 Names of MathML Encodings are the exact strings that should be used to identify the transfer flavors that correspond to the MathML encodings. On operating systems that allow such, an application should register their support for these flavor names (e.g. on Windows, a call to RegisterClipboardFormat, or, on the Macintosh platform, declaration of support for the universal type identifier in the application descriptor).

When transferring MathML, an application *MUST* ensure the content of the data transfer is a well-formed XML instance of a MathML document type. Specifically:

1. The instance *MAY* begin with an XML declaration, e.g. <?xml version="1.0">
2. The instance *MUST* contain exactly one root `math` element.
3. The instance *MUST* declare the MathML namespace on the root `math` element.
4. The instance *MAY* use a `schemaLocation` attribute on the `math` element to indicate the location of the MathML schema that describes the MathML document type to which the instance conforms. The presence of the `schemaLocation` attribute does not require a consumer of the MathML instance to obtain or use the referenced schema.
5. The instance *SHOULD* use numeric character references (e.g. α) rather than character entity names (e.g. α) for greater interoperability.
6. The instance *MUST* specify the character encoding, if it uses an encoding other than UTF-8, either in the XML declaration, or by the use of a byte-order mark (BOM) for UTF-16-encoded data.

7.3.2 Recommended Behaviors when Transferring

An application that transfers MathML markup *SHOULD* adhere to the following conventions:

1. An application that supports pure presentation markup and/or pure content markup *SHOULD* offer as many of these flavors as it has available.
2. An application that only exports one MathML flavor *SHOULD* name it `MathML` if it is unable to determine a more specific flavor.
3. If an application is able to determine a more specific flavor, it *SHOULD* offer both the generic and specific transfer flavors, but it *SHOULD* only deliver the specific flavor if it knows that the recipient supports it. For an HTTP GET transfer, for example, the specific transfer types for content and presentation markup should only be returned if they are included in the HTTP `Accept` header sent by the client.
4. An application that exports the two specific transfer flavors *SHOULD* export both the content and presentation transfer flavors, as well as the generic flavor, which *SHOULD* combine the other two flavors using a top-level MathML

`semantics` element (see 6.9.1 Top-level Parallel Markup).

5. When an application exports a MathML fragment whose only child of the root element is a `semantics` element, it *SHOULD* offer, after the above flavors, a transfer flavor for each `annotation` or `annotation-xml` element, provided the transfer flavor can be recognized and named based on the `encoding` attribute value, and provided the annotation key is (the default) alternate-representation. The transfer content for each annotation should contain the character data in the specified encoding (for an `annotation` element), or a well-formed XML fragment (for an `annotation-xml` element), or the data that results by requesting the URL given by the `src` attribute (for an annotation reference).

6. As a final fallback, an application *MAY* export a version of the data in a plain-text flavor (such as `text/plain`, `CF_UNICODETEXT`, `UnicodeText`, or `NSStringPboardType`). When an application has multiple versions of an expression available, it may choose the version to export as text at its discretion. Since some older MathML processors expect MathML instances transferred as plain text to begin with a `math` element, the text version *SHOULD* generally omit the XML declaration, DOCTYPE declaration, and other XML prolog material that would appear before the `math` element. The Unicode text version of the data *SHOULD* always be the last flavor exported, following the principle that exported flavors should be ordered with the most specific flavor first and the least specific flavor last.

7.3.3 Discussion

To determine whether a MathML instance is pure content markup or pure presentation markup, the `math`, `semantics`, `annotation` and `annotation-xml` elements should be regarded as belonging to both the presentation and content markup vocabularies. The `math` element is treated in this way because it is required as the root element in any MathML transfer. The `semantics` element and its child annotation elements comprise an arbitrary annotation mechanism within MathML, and are not tied to either presentation or content markup. Consequently, an application that consumes MathML should always process these four elements, even if it only implements one of the two vocabularies.

It is worth noting that the above recommendations allow agents that produce MathML to provide binary data for the clipboard, for example in an image or other application-specific format. The sole method to do so is to reference the binary data using the `src` attribute of an annotation, since XML character data does not allow for the transfer of arbitrary byte-stream data.

While the above recommendations are intended to improve interoperability between MathML-aware applications that use these transfer paradigms, it should be noted that they do not guarantee interoperability. For example, references to external resources (e.g. stylesheets, etc.) in MathML data can cause interoperability problems if the consumer of the data is unable to locate them, as can happen when cutting and pasting HTML or other data types. An application that makes use of references to external resources is encouraged to make users aware of potential problems and provide alternate ways to obtain the referenced resources. In general, consumers of MathML data that contains references they cannot resolve or do not understand should ignore the external references.

7.3.4 Examples

Example 1

An e-learning application has a database of quiz questions, some of which contain MathML. The MathML comes from multiple sources, and the e-learning application merely passes the data on for display, but does not have sophisticated MathML analysis capabilities. Consequently, the application is not aware whether a given MathML instance is pure presentation or pure content markup, nor does it know whether the instance is valid with respect to a particular version of the MathML schema. It therefore places the following data formats on the clipboard:

Flavor Name	Flavor Content
MathML	`<math xmlns="http://www.w3.org/1998/Math/MathML">...</math>`
Unicode Text	`<math xmlns="http://www.w3.org/1998/Math/MathML">...</math>`

Example 2

An equation editor on the Windows platform is able to generate pure presentation markup, valid with respect to MathML 3. Consequently, it exports the following flavors:

Flavor Name	Flavor Content
MathML Presentation	`<math xmlns="http://www.w3.org/1998/Math/MathML">...</math>`
Tiff	(a rendering sample)
Unicode Text	`<math xmlns="http://www.w3.org/1998/Math/MathML">...</math>`

Example 3

A schema-based content management system on the Mac OS X platform contains multiple MathML representations of a collection of mathematical expressions, including mixed markup from authors, pure content markup for interfacing to symbolic computation engines, and pure presentation markup for print publication. Due to the system's use of schemata, markup is stored with a namespace prefix. The system therefore can transfer the following data:

Flavor Name	Flavor Content
public.mathml.presentation	`<math xmlns="http://www.w3.org/1998/Math/MathML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation= "http://www.w3.org/Math/XMLSchema/mathml4/mathml4.xsd"> <mrow> ... </mrow> </math>`
public.mathml.content	`<math xmlns="http://www.w3.org/1998/Math/MathML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation= "http://www.w3.org/Math/XMLSchema/mathml4/mathml4.xsd"> <apply> ... </apply> </math>`
public.mathml	`<math xmlns="http://www.w3.org/1998/Math/MathML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation= "http://www.w3.org/Math/XMLSchema/mathml4/mathml4.xsd"> <mrow> <apply> ... content markup within presentation markup ... </apply> ... </mrow> </math>`
public.plain-text.tex	`{x \over x-1}`
	`<math xmlns="http://www.w3.org/1998/Math/MathML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=`

Example 4

A similar content management system is web-based and delivers MathML representations of mathematical expressions. The system is able to produce MathML-Presentation, MathML-Content, TeX and pictures in TIFF format. In web-pages being browsed, it could produce a MathML fragment such as the following:

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <semantics>
    <mrow>...</mrow>
    <annotation-xml encoding="MathML-Content">...</annotation-xml>
    <annotation encoding="TeX">{1 \over x}</annotation>
    <annotation encoding="image/tiff" src="formula3848.tiff"/>
  </semantics>
</math>
```

A web browser on the Windows platform that receives such a fragment and tries to export it as part of a drag-and-drop action can offer the following flavors:

Flavor Name	Flavor Content
MathML Presentation	`<math xmlns="http://www.w3.org/1998/Math/MathML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation= "http://www.w3.org/Math/XMLSchema/mathml4/mathml4.xsd"> <mrow> ... </mrow> </math>`
MathML Content	`<math xmlns="http://www.w3.org/1998/Math/MathML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation= "http://www.w3.org/Math/XMLSchema/mathml4/mathml4.xsd"> <apply> ... </apply> </math>`
MathML	`<math xmlns="http://www.w3.org/1998/Math/MathML" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation= "http://www.w3.org/Math/XMLSchema/mathml4/mathml4.xsd"> <mrow> <apply> ... content markup within presentation markup ... </apply> ... </mrow> </math>`
TeX	`{x \over x-1}`
CF_TIFF	(the content of the picture file, requested from formula3848.tiff)
	`<math xmlns="http://www.w3.org/1998/Math/MathML"`

7.4 Combining MathML and Other Formats

MathML is usually used in combination with other markup languages. The most typical case is perhaps the use of MathML within a document-level markup language, such as HTML or DocBook. It is also common that other object-level markup languages are also included in a compound document format, such as MathML and SVG in HTML5. Other common use cases include mixing other markup within MathML. For example, an authoring tool might insert an element representing a cursor position or other state information within MathML markup, so that an author can pick up editing where it was broken off.

Most document markup languages have some concept of an inline equation (or graphic, object, etc.), so there is typically a natural way to incorporate MathML instances into the content model. However, in the other direction, embedding of markup within MathML is not so clear cut, since in many MathML elements, the role of child elements is defined by position. For example, the first child of an `apply` must be an operator, and the second child of an `mfrac` is the denominator. The proper behavior when foreign markup appears in such contexts is problematic. Even when such behavior can be defined in a particular context, it presents an implementation challenge for generic MathML processors.

For this reason, the default MathML schema does not allow foreign markup elements to be included within MathML instances.

In the standard schema, elements from other namespaces are not allowed, but attributes from other namespaces are permitted. MathML processors that encounter unknown XML markup should behave as follows:

1. An attribute from a non-MathML namespace should be silently ignored.
2. An element from a non-MathML namespace should be treated as an error, except in an `annotation-xml` element. If the element is a child of a presentation element, it should be handled as described in 3.3.5 Error Message `<merror>`. If the element is a child of a content element, it should be handled as described in 4.2.9 Error Markup `<cerror>`.

For example, if the second child of an `mfrac` element is an unknown element, the fraction should be rendered with a denominator that indicates the error.

When designing a compound document format in which MathML is included in a larger document type, the designer may extend the content model of MathML to allow additional elements. For example, a common extension is to extend the MathML schema such that elements from non-MathML namespaces are allowed in token elements, but not in other elements. MathML processors that encounter unknown markup should behave as follows:

1. An unrecognized XML attribute should be silently ignored.
2. An unrecognized element in a MathML token element should be silently ignored.

3. An element from a non-MathML namespace should be treated as an error, except in an `annotation-xml` element. If the element is a child of a presentation element, it should be handled as described in 3.3.5 Error Message `<merror>`. If the element is a child of a content element, it should be handled as described in 4.2.9 Error Markup `<cerror>`.

Extending the schema in this way is easily achieved using the Relax NG schema described in A. Parsing MathML, it may be as simple as including the MathML schema whilst overriding the content model of `mtext`:

```
default namespace m = "http://www.w3.org/1998/Math/MathML"

include "mathml4.rnc" {
mtext = element mtext {mtext.attributes, (token.content|anyElement)*}
}
```

The definition given here would allow any well formed XML that is not in the MathML namespace as a child of `mtext`. In practice this may be too lax. For example, an XHTML+MathML Schema may just want to allow inline XHTML elements as additional children of `mtext`. This may be achieved by replacing `anyElement` by a suitable production from the schema for the host document type, see 7.4.1 Mixing MathML and XHTML.

Considerations about mixing markup vocabularies in compound documents arise when a compound document type is first designed. But once the document type is fixed, it is not generally practical for specific software tools to further modify the content model to suit their needs. However, it is still frequently the case that such tools may need to store additional information within a MathML instance. Since MathML is most often generated by authoring tools, a particularly common and important case is where an authoring tool needs to store information about its internal state along with a MathML expression, so an author can resume editing from a previous state. For example, placeholders may be used to indicate incomplete parts of an expression, or an insertion point within an expression may need to be stored.

An application that needs to persist private data within a MathML expression should generally attempt to do so without altering the underlying content model, even in situations where it is feasible to do so. To support this requirement, regardless of what may be allowed by the content model of a particular compound document format, MathML permits the storage of private data via the following strategies:

1. In a format that permits the use of XML Namespaces, for small amounts of data, attributes from other namespaces are allowed on all MathML elements.

2. For larger amounts of data, applications may use the `semantics` element, as described in 6. Annotating MathML: semantics.

3. For authoring tools and other applications that need to associate particular actions with presentation MathML subtrees, e.g. to mark an incomplete expression to be filled in by an author, the `maction` element may be used, as described in 3.7.1 Bind Action to Sub-Expression.

7.4.1 Mixing MathML and XHTML

To fully integrate MathML into XHTML, it should be possible not only to embed MathML in XHTML, but also to embed XHTML in MathML. The schema used for the W3C HTML5 validator extends `mtext` to allow all inline (phrasing) HTML elements (including `svg`) to be used within the content of `mtext`. See the example in 3.2.2.1 Embedding HTML in MathML. As noted above, MathML fragments using XHTML elements within `mtext` will not be valid MathML if extracted

from the document and used in isolation. Editing tools may offer support for removing any HTML markup from within `mtext` and replacing it by a text alternative.

In most cases, XHTML elements (headings, paragraphs, lists, etc.) either do not apply in mathematical contexts, or MathML already provides equivalent or improved functionality specifically tailored to mathematical content (tables, mathematics style changes, etc.).

Consult the W3C Math Working Group home page for compatibility and implementation suggestions for current browsers and other MathML-aware tools.

7.4.2 Mixing MathML and non-XML contexts

There may be non-XML vocabularies which require markup for mathematical expressions, where it makes sense to reference this specification. HTML is an important example discussed in the next section, however other examples exist. It is possible to use a TeX-like syntax such as `\frac{a}{b}` rather than explicitly using `<mfrac>` and `<mi>`. If a system parses a specified syntax and produces a tree that may be validated against the MathML schema then it may be viewed as a MathML application. Note however that documents using such a system are not valid MathML. Implementations of such a syntax should, if possible, offer a facility to output any mathematical expressions as MathML in the XML syntax defined here. Such an application would then be a MathML-output-conformant processor as described in D.1 MathML Conformance.

7.4.3 Mixing MathML and HTML

An important example of a non-XML based system is defined in [HTML]. When considering MathML in HTML there are two separate issues to consider. Firstly the schema is extended to allow HTML in `mtext` as described above in the context of XHTML. Secondly an HTML parser is used rather than an XML parser. The parsing of MathML by an HTML parser is normatively defined in [HTML]. The description there is aimed at parser implementers and written in terms of the state transitions of the parser as it parses each character of the input. The *non-normative* description below aims to give a higher level description and examples.

XML parsing is completely regular, any XML document may be parsed without reference to the particular vocabulary being used. HTML parsing differs in that it is a custom parser for the HTML vocabulary with specific rules for each element. Similarly to XML though, the HTML parser distinguishes parsing from validation; some input, even if it renders correctly, is classed as a *parse error* which may be reported by validators (but typically is not reported by rendering systems).

The main differences that affect MathML usage may be summarized as:

- Attribute values in most cases do not need to be quoted: `<mfenced open=( close=)>` would parse correctly.
- End tags may in many cases be omitted.
- HTML does not support namespaces other than the three built in ones for HTML, MathML and SVG, and does not support namespace prefixes. Thus you can not use a prefix form like `<mml:math xmlns:mml="http://www.w3.org/1998/Math/MathML">` and while you may use `<math xmlns="http://www.w3.org/1998/Math/MathML">`, the namespace declaration is essentially ignored and the input is treated as `<math>`. In either case the `math` element and its descendants are placed in the MathML namespace. As noted in 6. Annotating MathML: semantics the

lack of namespace support limits some of the possibilities for annotating MathML with markup from other vocabularies when used in HTML.

- Unlike the XML parser, the HTML parser is defined to accept *any* input string and produce a defined result (which may be classified as non-conforming). The extreme example `<math></><z =5>` for example would be flagged as a parse error by validators but would return a tree corresponding to a math element containing a comment `<` and an element `z` with an attribute that could not be expressed in XML with name `=5` and value `""`.
- Unless inside the token elements `<mtext>`, `<mo>`, `<mn>`, `<mi>`, `<ms>`, or inside an `<annotation-xml>` with `encoding` attribute `text/html` or `annotation/xhtml+xml`, the presence of an HTML element will *terminate* the math expression by closing all open MathML elements, so that the HTML element is interpreted as being in the outer HTML context. Any following MathML elements are then not contained in `<math>` so will be parsed as invalid HTML elements and not rendered as MathML. See for example the example given in 6.7.3 Using `annotation-xml` in HTML documents.

In the interests of compatibility with existing MathML applications authors and editing systems *should* use MathML fragments that are well formed XML, even when embedded in an HTML document. Also as noted above, although applications accepting MathML in HTML documents must accept MathML making use of these HTML parser features, they should offer a way to export MathML in a portable XML syntax.

7.4.4 Linking

In MathML 3, an element is designated as a link by the presence of the `href` attribute. MathML has no element that corresponds to the HTML/XHTML anchor element `a`.

MathML allows the `href` attribute on all elements. However, most user agents have no way to implement nested links or links on elements with no visible rendering; such links may have no effect.

The list of presentation markup elements that do not ordinarily have a visual rendering, and thus should not be used as linking elements, is given in the table below.

MathML elements that should not be linking elements

`mprescripts`	`none`
`malignmark`	`maligngroup`

For compound document formats that support linking mechanisms, the `id` attribute should be used to specify the location for a link into a MathML expression. The `id` attribute is allowed on all MathML elements, and its value must be unique within a document, making it ideal for this purpose.

Note that MathML 2 has no direct support for linking; it refers to the W3C Recommendation "XML Linking Language" [XLink] in defining links in compound document contexts by using an `xlink:href` attribute. As mentioned above, MathML 3 adds an `href` attribute for linking so that `xlink:href` is no longer needed. However, `xlink:href` is still allowed because MathML permits the use of attributes from non-MathML namespaces. It is recommended that new compound document formats use the MathML 3 `href` attribute for linking. When user agents encounter MathML elements with both `href` and `xlink:href` attributes, the `href` attribute should take precedence. To support backward compatibility, user agents that implement XML Linking in compound documents containing MathML 2 should continue to support the use of the `xlink:href` attribute in addition to supporting the `href` attribute.

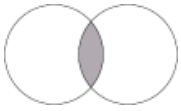
7.4.5 MathML and Graphical Markup

Apart from the introduction of new glyphs, many of the situations where one might be inclined to use an image amount to displaying labeled diagrams. For example, knot diagrams, Venn diagrams, Dynkin diagrams, Feynman diagrams and commutative diagrams all fall into this category. As such, their content would be better encoded via some combination of structured graphics and MathML markup. However, at the time of this writing, it is beyond the scope of the W3C Math Activity to define a markup language to encode such a general concept as "labeled diagrams." (See http://www.w3.org/Math for current W3C activity in mathematics and http://www.w3.org/Graphics for the W3C graphics activity.)

One mechanism for embedding additional graphical content is via the `semantics` element, as in the following example:

```
<semantics>
  <apply>
    <intersect/>
    <ci>A</ci>
    <ci>B</ci>
  </apply>
  <annotation-xml encoding="image/svg+xml">
    <svg xmlns="http://www.w3.org/2000/svg"  viewBox="0 0 290 180">
      <clipPath id="a">
        <circle cy="90" cx="100" r="60"/>
      </clipPath>
      <circle fill="#AAAAAA" cy="90" cx="190" r="60" style="clip-path:url(#a)"/>
      <circle stroke="black" fill="none" cy="90" cx="100" r="60"/>
      <circle stroke="black" fill="none" cy="90" cx="190" r="60"/>
    </svg>
  </annotation-xml>
  <annotation-xml encoding="application/xhtml+xml">
    <img xmlns="http://www.w3.org/1999/xhtml" src="intersect.png" alt="A intersect B"/>
  </annotation-xml>
</semantics>
```

Here, the `annotation-xml` elements are used to indicate alternative representations of the MathML-Content depiction of the intersection of two sets. The first one is in the "Scalable Vector Graphics" format [SVG] (see [XHTML-MathML-SVG] for the definition of an XHTML profile integrating MathML and SVG), the second one uses the XHTML `img` element embedded as an XHTML fragment. In this situation, a MathML processor can use any of these representations for display, perhaps producing a graphical format such as the image below.



Note that the semantics representation of this example is given in MathML-Content markup, as the first child of the `semantics` element. In this regard, it is the representation most analogous to the `alt` attribute of the `img` element in XHTML, and would likely be the best choice for non-visual rendering.

7.5 Using CSS with MathML

When MathML is rendered in an environment that supports CSS [CSS21], controlling mathematics style properties with a CSS style sheet is desirable, but not as simple as it might first appear, because the formatting of MathML layout schemata is quite different from the CSS visual formatting model and many of the style parameters that affect mathematics layout have no direct textual analogs. Even in cases where there are analogous properties, the sensible values for these properties may not correspond. Because of this difference, applications that support MathML natively may choose to restrict the CSS properties applicable to MathML layout schemata to those properties that do not affect layout.

Generally speaking, the model for CSS interaction with the math style attributes runs as follows. A CSS style sheet might provide a style rule such as:

```
math *.[mathsize="small"] {
font-size: 80%
}
```

This rule sets the CSS font-size property for all children of the `math` element that have the `mathsize` attribute set to small. A MathML renderer would then query the style engine for the CSS environment, and use the values returned as input to its own layout algorithms. MathML does not specify the mechanism by which style information is inherited from the environment. However, some suggested rendering rules for the interaction between properties of the ambient style environment and MathML-specific rendering rules are discussed in 3.2.2 Mathematics style attributes common to token elements, and more generally throughout 3. Presentation Markup.

It should be stressed, however, that some caution is required in writing CSS stylesheets for MathML. Because changing typographic properties of mathematics symbols can change the meaning of an equation, stylesheets should be written in a way such that changes to document-wide typographic styles do not affect embedded MathML expressions.

Another pitfall to be avoided is using CSS to provide typographic style information necessary to the proper understanding of an expression. Expressions dependent on CSS for meaning will not be portable to non-CSS environments such as computer algebra systems. By using the logical values of the new MathML 3.0 mathematics style attributes as selectors for CSS rules, it can be assured that style information necessary to the sense of an expression is encoded directly in the MathML.

MathML 3.0 does not specify how a user agent should process style information, because there are many non-CSS MathML environments, and because different users agents and renderers have widely varying degrees of access to CSS information.

7.5.1 Order of processing attributes versus style sheets

CSS or analogous style sheets can specify changes to rendering properties of selected MathML elements. Since rendering properties can also be changed by attributes on an element, or be changed automatically by the renderer, it is necessary to specify the order in which changes requested by various sources should occur. The order is defined by [CSS21] cascading order taking into account precedence of non-CSS presentational hints.

8. Characters, Entities and Fonts

ISSUE 247: Spec should specify what char to use for accents/lines **need specification update**

TeX has a number of commands that correspond to `mover`/`munder` accents in MathML. The spec does not say what character to use for those accents. In some cases there are ASCII chars that could be used but also non-ASCII ones that are similar. Many of these characters should be stretchy when used with `mover`/`munder`.

8.1 Introduction

This chapter contains discussion of characters for use within MathML, recommendations for their use, and warnings concerning the correct form of the corresponding code points given in the Universal Multiple-Octet Coded Character Set (UCS) ISO-10646 as codified in Unicode [Unicode].

8.2 Mathematical Alphanumeric Symbols

Additional Mathematical Alphanumeric Symbols were provided in Unicode 3.1. As discussed in 3.2.2 Mathematics style attributes common to token elements, MathML offers an alternative mechanism to specify mathematical alphanumeric characters. Namely, one uses the `mathvariant` attribute on a token element such as `mi` to indicate that the character data in the token element selects a mathematical alphanumeric symbol.

An important use of the mathematical alphanumeric symbols in Plane 1 is for identifiers normally printed in special mathematical fonts, such as Fraktur, Greek, Boldface, or Script. As another example, the Mathematical Fraktur alphabet runs from U+1D504 ("A") to U+1D537 ("z"). Thus, an identifier for a variable that uses Fraktur characters could be marked up as

```
<mi>&#x1D504;<!--BLACK-LETTER CAPITAL A--></mi>
```

𝔄

An alternative, equivalent markup for this example is to use the common upper-case A, modified by using the `mathvariant` attribute:

```
<mi mathvariant="fraktur">A</mi>
```

𝔄

A MathML processor must treat a mathematical alphanumeric character (when it appears) as identical to the corresponding combination of the unstyled character and mathvariant attribute value.

It is intended that renderers distinguish at least those combinations that have equivalent Unicode code points, and renderers are free to ignore those combinations that have no assigned Unicode code point or for which adequate font support is unavailable.

8.3 Non-Marking Characters

Some characters, although important for the quality of print or alternative rendering, do not have glyph marks that correspond directly to them. They are called here non-marking characters. Their roles are discussed in 3. Presentation Markup and 4. Content Markup.

In MathML, control of page composition, such as line-breaking, is effected by the use of the proper attributes on the `mo` and `mspace` elements.

The characters below are not simple spacers. They are especially important new additions to the UCS because they provide textual clues which can increase the quality of print rendering, permit correct audio rendering, and allow the unique recovery of mathematical semantics from text which is visually ambiguous.

Unicode code point	Unicode name	Description
U+2061	FUNCTION APPLICATION	character showing function application in presentation tagging (3.2.5 Operator, Fence, Separator or Accent `<mo>`)
U+2062	INVISIBLE TIMES	marks multiplication when it is understood without a mark (3.2.5 Operator, Fence, Separator or Accent `<mo>`)
U+2063	INVISIBLE SEPARATOR	used as a separator, e.g., in indices (3.2.5 Operator, Fence, Separator or Accent `<mo>`)
U+2064	INVISIBLE PLUS	marks addition, especially in constructs such as 1½ (3.2.5 Operator, Fence, Separator or Accent `<mo>`)

8.4 Anomalous Mathematical Characters

Some characters which occur fairly often in mathematical texts, and have special significance there, are frequently confused with other similar characters in the UCS. In some cases, common keyboard characters have become entrenched as alternatives to the more appropriate mathematical characters. In others, characters have legitimate uses in both formulas and text, but conflicting rendering and font conventions. All these characters are called here anomalous characters.

8.4.1 Keyboard Characters

Typical Latin-1-based keyboards contain several characters that are visually similar to important mathematical characters. Consequently, these characters are frequently substituted, intentionally or unintentionally, for their more correct mathematical counterparts.

Minus

The most common ordinary text character which enjoys a special mathematical use is U+002D [HYPHEN-MINUS]. As its Unicode name suggests, it is used as a hyphen in prose contexts, and as a minus or negative sign in formulas. For text use, there is a specific code point U+2010 [HYPHEN] which is intended for prose contexts, and which should render as a hyphen or short dash. For mathematical use, there is another code point U+2212 [MINUS SIGN] which is intended for mathematical formulas, and which should render as a longer minus or negative sign. MathML renderers should treat U+002D [HYPHEN-MINUS] as equivalent to U+2212 [MINUS SIGN] in formula contexts such as `mo`, and as equivalent to U+2010 [HYPHEN] in text contexts such as `mtext`.

Apostrophes, Quotes and Primes

On a typical European keyboard there is a key available which is viewed as an apostrophe or a single quotation mark (an upright or right quotation mark). Thus one key is doing double duty for prose input to enter U+0027 [APOSTROPHE] and U+2019 [RIGHT SINGLE QUOTATION MARK]. In mathematical contexts it is also commonly used for the prime, which should be U+2032 [PRIME]. Unicode recognizes the overloading of this symbol and remarks that it can also signify the units of minutes or feet. In the unstructured printed text of normal prose the characters are placed next to one another. The U+0027 [APOSTROPHE] and U+2019 [RIGHT SINGLE QUOTATION MARK] are marked with glyphs that are small and raised with respect to the center line of the text. The fonts used provide small raised glyphs in the appropriate places indexed by the Unicode codes. The U+2032 [PRIME] of mathematics is similarly treated in fuller Unicode fonts.

MathML renderers are encouraged to treat U+0027 [APOSTROPHE] as U+2032 [PRIME] when appropriate in formula contexts.

A final remark is that a ‘prime’ is often used in transliteration of the Cyrillic character U+044C [CYRILLIC SMALL LETTER SOFT SIGN]. This different use of primes is not part of considerations for mathematical formulas.

Other Keyboard Substitutions

While the minus and prime characters are the most common and important keyboard characters with more precise mathematical counterparts, there are a number of other keyboard character substitutions that are sometimes used. For example some may expect

```
<mo>''</mo>
```

″

to be treated as U+2033 [DOUBLE PRIME], and analogous substitutions could perhaps be made for U+2034 [TRIPLE PRIME] and U+2057 [QUADRUPLE PRIME]. Similarly, sometimes U+007C [VERTICAL LINE] is used for U+2223 [DIVIDES]. MathML regards these as application-specific authoring conventions, and recommends that authoring tools generate markup using the more precise mathematical characters for better interoperability.

8.4.2 Pseudo-scripts

There are a number of characters in the UCS that traditionally have been taken to have a natural ‘script’ aspect. The visual presentation of these characters is similar to a script, that is, raised from the baseline, and smaller than the base font size. The degree symbol and prime characters are examples. For use in text, such characters occur in sequence with the identifier they follow, and are typically rendered using the same font. These characters are called pseudo-scripts here.

In almost all mathematical contexts, pseudo-script characters should be associated with a base expression using explicit script markup in MathML. For example, the preferred encoding of “x prime” is

```
<msup><mi>x</mi><mo>&#x2032;<!--PRIME--></mo></msup>
```

$$x'$$

and not

```
<mi>x'</mi>
```

$$x'$$

or any other variants not using an explicit script construct. Note, however, that within text contexts such as `mtext`, pseudo-scripts may be used in sequence with other character data.

There are two reasons why explicit markup is preferable in mathematical contexts. First, a problem arises with typesetting, when pseudo-scripts are used with subscripted identifiers. Traditionally, subscripting of x' would be rendered stacked under the prime. This is easily accomplished with script markup, for example:

```
<mrow><msubsup><mi>x</mi><mn>0</mn><mo>&#x2032;<!--PRIME--></mo></msubsup></mrow>
```

$$x'_0$$

By contrast,

```
<mrow><msub><mi>x'</mi><mn>0</mn></msub></mrow>
```

$$x'_0$$

will render with staggered scripts.

Note this means that a renderer of MathML will have to treat pseudo-scripts differently from most other character codes it

finds in a superscript position; in most fonts, the glyphs for pseudo-scripts are already shrunk and raised from the baseline.

The second reason that explicit script markup is preferrable to juxtaposition of characters is that it generally better reflects the intended mathematical structure. For example,

```
<msup>
  <mrow><mo>(</mo><mrow><mi>f</mi><mo>+</mo><mi>g</mi></mrow><mo>)</mo></mrow>
  <mo>&#x2032;<!--PRIME--></mo>
</msup>
```

$$(f+g)'$$

accurately reflects that the prime here is operating on an entire expression, and does not suggest that the prime is acting on the final right parenthesis.

However, the data model for all MathML token elements is Unicode text, so one cannot rule out the possibility of valid MathML markup containing constructions such as

```
<mrow><mi>x'</mi></mrow>
```

x'

and

```
<mrow><mi>x</mi><mo>'</mo></mrow>
```

x'

While the first form may, in some rare situations, legitimately be used to distinguish a multi-character identifer named x' from the derivative of a function x, such forms should generally be avoided. Authoring and validation tools are encouraged to generate the recommended script markup:

```
<mrow><msup><mi>x</mi><mo>&#x2032;<!--PRIME--></mo></msup></mrow>
```

x'

The U+2032 [PRIME] character is perhaps the most common pseudo-script, but there are many others, as listed below:

Pseudo-script Characters
U+0022 QUOTATION MARK
U+0027 APOSTROPHE
U+002A ASTERISK

Pseudo-script Characters

U+0060 GRAVE ACCENT

U+00AA FEMININE ORDINAL INDICATOR

U+00B0 DEGREE SIGN

U+00B2 SUPERSCRIPT TWO

U+00B3 SUPERSCRIPT THREE

U+00B4 ACUTE ACCENT

U+00B9 SUPERSCRIPT ONE

U+00BA MASCULINE ORDINAL INDICATOR

U+2018 LEFT SINGLE QUOTATION MARK

U+2019 RIGHT SINGLE QUOTATION MARK

U+201A SINGLE LOW-9 QUOTATION MARK

U+201B SINGLE HIGH-REVERSED-9 QUOTATION MARK

U+201C LEFT DOUBLE QUOTATION MARK

U+201D RIGHT DOUBLE QUOTATION MARK

U+201E DOUBLE LOW-9 QUOTATION MARK

U+201F DOUBLE HIGH-REVERSED-9 QUOTATION MARK

U+2032 PRIME

U+2033 DOUBLE PRIME

U+2034 TRIPLE PRIME

U+2035 REVERSED PRIME

U+2036 REVERSED DOUBLE PRIME

U+2037 REVERSED TRIPLE PRIME

U+2057 QUADRUPLE PRIME

In addition, the characters in the Unicode Superscript and Subscript block (beginning at U+2070) should be treated as pseudo-scripts when they appear in mathematical formulas.

Note that several of these characters are common on keyboards, including U+002A [ASTERISK], U+00B0 [DEGREE SIGN], U+2033 [DOUBLE PRIME], and U+2035 [REVERSED PRIME] also known as a back prime.

8.4.3 Combining Characters

In the UCS there are many combining characters that are intended to be used for the many accents of numerous different natural languages. Some of them may seem to provide markup needed for mathematical accents. They should not be used in mathematical markup. Superscript, subscript, underscript, and overscript constructions as just discussed above should be used for this purpose. Of course, combining characters may be used in multi-character identifiers as they are needed, or in text contexts.

There is one more case where combining characters turn up naturally in mathematical markup. Some relations have associated negations, such as U+226F [NOT GREATER-THAN] for the negation of U+003E [GREATER-THAN SIGN]. The glyph for U+226F [NOT GREATER-THAN] is usually just that for U+003E [GREATER-THAN SIGN] with a slash

through it. Thus it could also be expressed by U+003E-0338 making use of the combining slash U+0338 [COMBINING LONG SOLIDUS OVERLAY]. That is true of 25 other characters in common enough mathematical use to merit their own Unicode code points. In the other direction there are 31 character entity names listed in [Entities] which are to be expressed using U+0338 [COMBINING LONG SOLIDUS OVERLAY].

In a similar way there are mathematical characters which have negations given by a vertical bar overlay U+20D2 [COMBINING LONG VERTICAL LINE OVERLAY]. Some are available in pre-composed forms, and some named character entities are given explicitly as combinations. In addition there are examples using U+0333 [COMBINING DOUBLE LOW LINE] and U+20E5 [COMBINING REVERSE SOLIDUS OVERLAY], and variants specified by use of the U+FE00 [VARIATION SELECTOR-1]. For fuller listing of these cases see the listings in [Entities].

The general rule is that a base character followed by a string of combining characters should be treated just as though it were the pre-composed character that results from the combination, if such a character exists.

A. Parsing MathML

ISSUE 178: Make MathML attributes ASCII case-insensitive **MathML 4** **css / html5**

Issue 178

~~ISSUE 361 (CLOSED)~~: structuring common attributes **MathML 4** **need specification update**

Issue 361

A.1 Validating MathML

The Relax NG schema may be used to check the XML serialization of MathML and serves as a foundation for validating other serializations of MathML, such as the HTML serialization.

Even when using the XML serialization, some normalization of the input may be required before applying this schema. Notably, following HTML, [MathML-Core] allows attributes such as `onclick` to be specified in any case, eg `OnClick="..."`. It is not practically feasible to specify that attribute names are case insensitive here so only the lowercase names are allowed. Similarly any attribute with name starting with the prefix `data-` should be considered valid. The schema here only allows a fixed attribute, `data-other`, so input should be normalized to remove data attributes before validating, or the schema should be extended to support the attributes used in a particular application.

A.2 Using the RelaxNG Schema for MathML

MathML documents should be validated using the RelaxNG Schema for MathML, either in the XML encoding (http://www.w3.org/Math/RelaxNG/mathml4/mathml4.rng) or in compact notation (https://www.w3.org/Math/RelaxNG/mathml4/mathml4.rnc) which is also shown below.

In contrast to DTDs there is no in-document method to associate a RelaxNG schema with a document.

A.2.1 MathML Core

MathML Core is specified in *MathML Core* however the Schema is developed alongside the schema for MathML 4 and presented here, it can also be found at https://www.w3.org/Math/RelaxNG/mathml4/mathml4-core.rnc.

```
# MathML 4 (Core Level 1)
# ######################

#     Copyright 1998-2024 W3C (MIT, ERCIM, Keio, Beihang)
#
#     Use and distribution of this code are permitted under the terms
#     W3C Software Notice and License
#     http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

default namespace m = "http://www.w3.org/1998/Math/MathML"
namespace h = "http://www.w3.org/1999/xhtml"

start |= math

math = element math {math.attributes,ImpliedMrow}

MathMLoneventAttributes =
  attribute onabort {text}?,
  attribute onauxclick {text}?,
  attribute onblur {text}?,
  attribute oncancel {text}?,
  attribute oncanplay {text}?,
  attribute oncanplaythrough {text}?,
  attribute onchange {text}?,
  attribute onclick {text}?,
  attribute onclose {text}?,
  attribute oncontextlost {text}?,
  attribute oncontextmenu {text}?,
  attribute oncontextrestored {text}?,
  attribute oncuechange {text}?,
  attribute ondblclick {text}?,
  attribute ondrag {text}?,
  attribute ondragend {text}?,
  attribute ondragenter {text}?,
  attribute ondragleave {text}?,
  attribute ondragover {text}?,
  attribute ondragstart {text}?,
  attribute ondrop {text}?,
  attribute ondurationchange {text}?,
  attribute onemptied {text}?,
  attribute onended {text}?,
  attribute onerror {text}?,
  attribute onfocus {text}?,
```

```
attribute onformdata {text}?,
attribute oninput {text}?,
attribute oninvalid {text}?,
attribute onkeydown {text}?,
attribute onkeypress {text}?,
attribute onkeyup {text}?,
attribute onload {text}?,
attribute onloadeddata {text}?,
attribute onloadedmetadata {text}?,
attribute onloadstart {text}?,
attribute onmousedown {text}?,
attribute onmouseenter {text}?,
attribute onmouseleave {text}?,
attribute onmousemove {text}?,
attribute onmouseout {text}?,
attribute onmouseover {text}?,
attribute onmouseup {text}?,
attribute onpause {text}?,
attribute onplay {text}?,
attribute onplaying {text}?,
attribute onprogress {text}?,
attribute onratechange {text}?,
attribute onreset {text}?,
attribute onresize {text}?,
attribute onscroll {text}?,
attribute onsecuritypolicyviolation {text}?,
attribute onseeked {text}?,
attribute onseeking {text}?,
attribute onselect {text}?,
attribute onslotchange {text}?,
attribute onstalled {text}?,
attribute onsubmit {text}?,
attribute onsuspend {text}?,
attribute ontimeupdate {text}?,
attribute ontoggle {text}?,
attribute onvolumechange {text}?,
attribute onwaiting {text}?,
attribute onwebkitanimationend {text}?,
attribute onwebkitanimationiteration {text}?,
attribute onwebkitanimationstart {text}?,
attribute onwebkittransitionend {text}?,
attribute onwheel {text}?,
attribute onafterprint {text}?,
attribute onbeforeprint {text}?,
attribute onbeforeunload {text}?,
attribute onhashchange {text}?,
attribute onlanguagechange {text}?,
attribute onmessage {text}?,
attribute onmessageerror {text}?,
attribute onoffline {text}?,
attribute ononline {text}?,
attribute onpagehide {text}?,
attribute onpageshow {text}?,
```

```
  attribute onpopstate {text}?,
  attribute onrejectionhandled {text}?,
  attribute onstorage {text}?,
  attribute onunhandledrejection {text}?,
  attribute onunload {text}?,
  attribute oncopy {text}?,
  attribute oncut {text}?,
  attribute onpaste {text}?

# Sample set. May need preprocessing
# or schema extension to allow more see MathML Core (and HTML) spec
MathMLDataAttributes =
  attribute data-other {text}?


# sample set, like data- may need preprocessing to allow more
MathMLARIAattributes =
  attribute aria-label {text}?,
  attribute aria-describedby {text}?,
  attribute aria-details {text}?

MathMLIntentAttributes =
  attribute intent {text}?,
  attribute arg {xsd:NCName}?

MathMLPGlobalAttributes = attribute id {xsd:ID}?,
                    attribute class {xsd:NCName}?,
                    attribute style {xsd:string}?,
                    attribute dir {"ltr" | "rtl"}?,
                    attribute mathbackground {color}?,
                    attribute mathcolor {color}?,
                    attribute mathsize {length-percentage}?,
                    attribute mathvariant {xsd:string{pattern="\s*([Nn][Oo][Rr][Mm][Aa]
[Ll]|[Bb][Oo][Ll][Dd]|[Ii][Tt][Aa][Ll][Ii][Cc]|[Bb][Oo][Ll][Dd]-[Ii][Tt][Aa][Ll][Ii][Cc]|
[Dd][Oo][Uu][Bb][Ll][Ee]-[Ss][Tt][Rr][Uu][Cc][Kk]|[Bb][Oo][Ll][Dd]-[Ff][Rr][Aa][Kk][Tt]
[Uu][Rr]|[Ss][Cc][Rr][Ii][Pp][Tt]|[Bb][Oo][Ll][Dd]-[Ss][Cc][Rr][Ii][Pp][Tt]|[Ff][Rr][Aa]
[Kk][Tt][Uu][Rr]|[Ss][Aa][Nn][Ss]-[Ss][Ee][Rr][Ii][Ff]|[Bb][Oo][Ll][Dd]-[Ss][Aa][Nn][Ss]-
[Ss][Ee][Rr][Ii][Ff]|[Ss][Aa][Nn][Ss]-[Ss][Ee][Rr][Ii][Ff]-[Ii][Tt][Aa][Ll][Ii][Cc]|[Ss]
[Aa][Nn][Ss]-[Ss][Ee][Rr][Ii][Ff]-[Bb][Oo][Ll][Dd]-[Ii][Tt][Aa][Ll][Ii][Cc]|[Mm][Oo][Nn]
[Oo][Ss][Pp][Aa][Cc][Ee]|[Ii][Nn][Ii][Tt][Ii][Aa][Ll]|[Tt][Aa][Ii][Ll][Ee][Dd]|[Ll][Oo]
[Oo][Pp][Ee][Dd]|[Ss][Tt][Rr][Ee][Tt][Cc][Hh][Ee][Dd])\s*"}}?,
                    attribute displaystyle {mathml-boolean}?,
                    attribute scriptlevel {xsd:integer}?,
                    attribute autofocus {mathml-boolean}?,
                    attribute tabindex {xsd:integer}?,
                    attribute nonce {text}?,
                    MathMLoneventAttributes,
                    # Extension attributes, no defined behavior
                    MathMLDataAttributes,
                    # No specified behavior in Core, see MathML4
                    MathMLIntentAttributes,
                    # No specified behavior in Core, see WAI-ARIA
                    MathMLARIAattributes
```

```
math.attributes = MathMLPGlobalAttributes,
                  attribute display {"block" | "inline"}?,
                  # No specified behavior in Core, see MathML4
                  attribute alttext {text}?


annotation = element annotation {MathMLPGlobalAttributes,encoding?,text}

anyElement =  element (*) {(attribute * {text}|text| anyElement)*}

annotation-xml = element annotation-xml {annotation-xml.attributes,
                                         (MathExpression*|anyElement*)}

annotation-xml.attributes = MathMLPGlobalAttributes, encoding?

encoding=attribute encoding {xsd:string}?


semantics = element semantics {semantics.attributes,
                               MathExpression,
                              (annotation|annotation-xml)*}

semantics.attributes = MathMLPGlobalAttributes

mathml-boolean = xsd:string {
    pattern = '\s*([Tt][Rr][Uu][Ee]|[Ff][Aa][Ll][Ss][Ee])\s*'
}

length-percentage = xsd:string {
  pattern = '\s*((-?[0-9]*([0-9]\.?|\.[0-9])[0-9]*(r?em|ex|in|cm|mm|p[xtc]|Q|v[hw]|vmin|
vmax|%))|0)\s*'
}

MathExpression = TokenExpression|
                     mrow|mfrac|msqrt|mroot|mstyle|merror|mpadded|mphantom|
                     msub|msup|msubsup|munder|mover|munderover|
                     mmultiscripts|mtable|maction|
                     semantics

MathMalignExpression = MathExpression

ImpliedMrow = MathMalignExpression*

TableRowExpression = mtr

MultiScriptExpression = (MathExpression|none),(MathExpression|none)


color =  xsd:string {
  pattern = '\s*((#[0-9a-fA-F]{3}([0-9a-fA-F]{3})?)|[a-zA-Z]+|[a-zA-Z]+\s*\([0-9, %.]+
```

```
\))\s*'}

TokenExpression = mi|mn|mo|mtext|mspace|ms


textorHTML = text | element (h:*)  {attribute * {text}*,textorHTML*}

token.content = textorHTML

mi = element mi {mi.attributes, token.content}
mi.attributes =
  MathMLPGlobalAttributes

mn = element mn {mn.attributes, token.content}
mn.attributes =
  MathMLPGlobalAttributes

mo = element mo {mo.attributes, token.content}
mo.attributes =
  MathMLPGlobalAttributes,
  attribute form {"prefix" | "infix" | "postfix"}?,
  attribute lspace {length-percentage}?,
  attribute rspace {length-percentage}?,
  attribute stretchy {mathml-boolean}?,
  attribute symmetric {mathml-boolean}?,
  attribute maxsize {length-percentage}?,
  attribute minsize {length-percentage}?,
  attribute largeop {mathml-boolean}?,
  attribute movablelimits {mathml-boolean}?


mtext = element mtext {mtext.attributes, token.content}
mtext.attributes =
  MathMLPGlobalAttributes

mspace = element mspace {mspace.attributes, empty}
mspace.attributes =
  MathMLPGlobalAttributes,
  attribute width {length-percentage}?,
  attribute height {length-percentage}?,
  attribute depth {length-percentage}?

ms = element ms {ms.attributes, token.content}
ms.attributes =
  MathMLPGlobalAttributes


none = element none {none.attributes,empty}
none.attributes =
  MathMLPGlobalAttributes

mprescripts = element mprescripts {mprescripts.attributes,empty}
mprescripts.attributes =
```

```
  MathMLPGlobalAttributes

mrow = element mrow {mrow.attributes, ImpliedMrow}
mrow.attributes =
  MathMLPGlobalAttributes

mfrac = element mfrac {mfrac.attributes, MathExpression, MathExpression}
mfrac.attributes =
  MathMLPGlobalAttributes,
  attribute linethickness {length-percentage}?

msqrt = element msqrt {msqrt.attributes, ImpliedMrow}
msqrt.attributes =
  MathMLPGlobalAttributes

mroot = element mroot {mroot.attributes, MathExpression, MathExpression}
mroot.attributes =
  MathMLPGlobalAttributes

mstyle = element mstyle {mstyle.attributes, ImpliedMrow}
mstyle.attributes =
  MathMLPGlobalAttributes

merror = element merror {merror.attributes, ImpliedMrow}
merror.attributes =
  MathMLPGlobalAttributes

mpadded = element mpadded {mpadded.attributes, ImpliedMrow}
mpadded.attributes =
  MathMLPGlobalAttributes,
  attribute height {mpadded-length-percentage}?,
  attribute depth {mpadded-length-percentage}?,
  attribute width {mpadded-length-percentage}?,
  attribute lspace {mpadded-length-percentage}?,
  attribute rspace {mpadded-length-percentage}?,
  attribute voffset {mpadded-length-percentage}?

mpadded-length-percentage=length-percentage

mphantom = element mphantom {mphantom.attributes, ImpliedMrow}
mphantom.attributes =
  MathMLPGlobalAttributes


msub = element msub {msub.attributes, MathExpression, MathExpression}
msub.attributes =
  MathMLPGlobalAttributes

msup = element msup {msup.attributes, MathExpression, MathExpression}
msup.attributes =
  MathMLPGlobalAttributes

msubsup = element msubsup {msubsup.attributes, MathExpression, MathExpression,
```

```
MathExpression}
msubsup.attributes =
  MathMLPGlobalAttributes

munder = element munder {munder.attributes, MathExpression, MathExpression}
munder.attributes =
  MathMLPGlobalAttributes,
  attribute accentunder {mathml-boolean}?

mover = element mover {mover.attributes, MathExpression, MathExpression}
mover.attributes =
  MathMLPGlobalAttributes,
  attribute accent {mathml-boolean}?

munderover = element munderover {munderover.attributes, MathExpression, MathExpression,
MathExpression}
munderover.attributes =
  MathMLPGlobalAttributes,
  attribute accent {mathml-boolean}?,
  attribute accentunder {mathml-boolean}?

mmultiscripts = element mmultiscripts {mmultiscripts.attributes,
                                       MathExpression,
                                       MultiScriptExpression*,
                                       (mprescripts,MultiScriptExpression*)?}
mmultiscripts.attributes =
  msubsup.attributes

mtable = element mtable {mtable.attributes, TableRowExpression*}
mtable.attributes =
  MathMLPGlobalAttributes


mtr = element mtr {mtr.attributes, mtd*}
mtr.attributes =
  MathMLPGlobalAttributes

mtd = element mtd {mtd.attributes, ImpliedMrow}
mtd.attributes =
  MathMLPGlobalAttributes,
  attribute rowspan {xsd:positiveInteger}?,
  attribute columnspan {xsd:positiveInteger}?


maction = element maction {maction.attributes, ImpliedMrow}
maction.attributes =
  MathMLPGlobalAttributes,
  attribute actiontype {text}?,
  attribute selection {xsd:positiveInteger}?
```

A.2.2 Presentation MathML

The grammar for Presentation MathML 4 builds on the grammar for the MathML Core, and can be found at https://www.w3.org/Math/RelaxNG/mathml4/mathml4-presentation.rnc.

```
# MathML 4 (Presentation)
# ######################

#     Copyright 1998-2024 W3C (MIT, ERCIM, Keio, Beihang)
#
#     Use and distribution of this code are permitted under the terms
#     W3C Software Notice and License
#     http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

default namespace m = "http://www.w3.org/1998/Math/MathML"
namespace local = ""


# MathML Core
include "mathml4-core.rnc" {

# named lengths
length-percentage = xsd:string {
  pattern = '\s*((-?[0-9]*([0-9]\.?|\.[0-9])[0-9]*(r?em|ex|in|cm|mm|p[xtc]|Q|v[hw]|vmin|
vmax|%))|0|(negative)?((very){0,2}thi(n|ck)|medium)mathspace)\s*'
}

mpadded-length-percentage = xsd:string {
  pattern = '\s*([\+\-]?[0-9]*([0-9]\.?|\.[0-9])[0-9]*\s*((%?\s*(height|depth|width)?)|r?
em|ex|in|cm|mm|p[xtc]|Q|v[hw]|vmin|vmax|%|((negative)?((very){0,2}thi(n|ck)|
medium)mathspace))?)\s*'
}


}

NonMathMLAtt = attribute (* - (local:* | m:*)) {xsd:string}

MathMLPGlobalAttributes &=
                    NonMathMLAtt*,
                    attribute xref {text}?,
                    attribute href {xsd:anyURI}?

MathMalignExpression |= MalignExpression

MathExpression |= PresentationExpression

MstackExpression = MathMalignExpression|mscarries|msline|msrow|msgroup

MsrowExpression = MathMalignExpression|none
```

```
linestyle = "none" | "solid" | "dashed"

verticalalign =
      "top" |
      "bottom" |
      "center" |
      "baseline" |
      "axis"

columnalignstyle = "left" | "center" | "right"

notationstyle =
     "longdiv" |
     "actuarial" |
     "radical" |
     "box" |
     "roundedbox" |
     "circle" |
     "left" |
     "right" |
     "top" |
     "bottom" |
     "updiagonalstrike" |
     "downdiagonalstrike" |
     "verticalstrike" |
     "horizontalstrike" |
     "madruwb"

idref = text
unsigned-integer = xsd:unsignedLong
integer = xsd:integer
number = xsd:decimal

character = xsd:string {
  pattern = '\s*\S\s*'}


positive-integer = xsd:positiveInteger


token.content |= mglyph|text



mo.attributes &=
  attribute linebreak {"auto" | "newline" | "nobreak" | "goodbreak" | "badbreak"}?,
  attribute lineleading {length-percentage}?,
  attribute linebreakstyle {"before" | "after" | "duplicate" | "infixlinebreakstyle"}?,
  attribute linebreakmultchar {text}?,
  attribute indentalign {"left" | "center" | "right" | "auto" | "id"}?,
  attribute indentshift {length-percentage}?,
```

```
  attribute indenttarget {idref}?,
  attribute indentalignfirst {"left" | "center" | "right" | "auto" | "id" |
"indentalign"}?,
  attribute indentshiftfirst {length-percentage | "indentshift"}?,
  attribute indentalignlast {"left" | "center" | "right" | "auto" | "id" |
"indentalign"}?,
  attribute indentshiftlast {length-percentage | "indentshift"}?,
  attribute accent {mathml-boolean}?,
  attribute maxsize {"infinity"}?


mspace.attributes &=
  attribute linebreak {"auto" | "newline" | "nobreak" | "goodbreak" | "badbreak" |
"indentingnewline"}?,
  attribute indentalign {"left" | "center" | "right" | "auto" | "id"}?,
  attribute indentshift {length-percentage}?,
  attribute indenttarget {idref}?,
  attribute indentalignfirst {"left" | "center" | "right" | "auto" | "id" |
"indentalign"}?,
  attribute indentshiftfirst {length-percentage | "indentshift"}?,
  attribute indentalignlast {"left" | "center" | "right" | "auto" | "id" |
"indentalign"}?,
  attribute indentshiftlast {length-percentage | "indentshift"}?


ms.attributes &=
  attribute lquote {text}?,
  attribute rquote {text}?


mglyph = element mglyph {mglyph.attributes,empty}
mglyph.attributes =
  MathMLPGlobalAttributes,
  attribute src {xsd:anyURI}?,
  attribute width {length-percentage}?,
  attribute height {length-percentage}?,
  attribute valign {length-percentage}?,
  attribute alt {text}?

msline = element msline {msline.attributes,empty}
msline.attributes =
  MathMLPGlobalAttributes,
  attribute position {integer}?,
  attribute length {unsigned-integer}?,
  attribute leftoverhang {length-percentage}?,
  attribute rightoverhang {length-percentage}?,
  attribute mslinethickness {length-percentage | "thin" | "medium" | "thick"}?

MalignExpression = maligngroup|malignmark

malignmark = element malignmark {malignmark.attributes, empty}
malignmark.attributes =  MathMLPGlobalAttributes
```

```
malign.group = element malign.group {malign.group.attributes, empty}
malign.group.attributes = MathMLPGlobalAttributes


PresentationExpression = TokenExpression|
                         mrow|mfrac|msqrt|mroot|mstyle|merror|mpadded|mphantom|
                         mfenced|menclose|msub|msup|msubsup|munder|mover|munderover|
                         mmultiscripts|mtable|mstack|mlongdiv|maction




mfrac.attributes &=
  attribute numalign {"left" | "center" | "right"}?,
  attribute denomalign {"left" | "center" | "right"}?,
  attribute bevelled {mathml-boolean}?



mstyle.attributes &=
  mstyle.specificattributes,
  mstyle.generalattributes

mstyle.specificattributes =
  attribute scriptsizemultiplier {number}?,
  attribute scriptminsize {length-percentage}?,
  attribute infixlinebreakstyle {"before" | "after" | "duplicate"}?,
  attribute decimalpoint {character}?

mstyle.generalattributes =
  attribute accent {mathml-boolean}?,
  attribute accentunder {mathml-boolean}?,
  attribute align {"left" | "right" | "center"}?,
  attribute alignmentscope {list {("true" | "false") +}}?,
  attribute bevelled {mathml-boolean}?,
  attribute charalign {"left" | "center" | "right"}?,
  attribute charspacing {length-percentage | "loose" | "medium" | "tight"}?,
  attribute close {text}?,
  attribute columnalign {list {columnalignstyle+} }?,
  attribute columnlines {list {linestyle +}}?,
  attribute columnspacing {list {(length-percentage) +}}?,
  attribute columnspan {positive-integer}?,
  attribute columnwidth {list {("auto" | length-percentage | "fit") +}}?,
  attribute crossout {list {("none" | "updiagonalstrike" | "downdiagonalstrike" |
"verticalstrike" | "horizontalstrike")*}}?,
  attribute denomalign {"left" | "center" | "right"}?,
  attribute depth {length-percentage}?,
  attribute dir {"ltr" | "rtl"}?,
  attribute equalcolumns {mathml-boolean}?,
  attribute equalrows {mathml-boolean}?,
  attribute form {"prefix" | "infix" | "postfix"}?,
```

```
  attribute frame {linestyle}?,
  attribute framespacing {list {length-percentage, length-percentage}}?,
  attribute height {length-percentage}?,
  attribute indentalign {"left" | "center" | "right" | "auto" | "id"}?,
  attribute indentalignfirst {"left" | "center" | "right" | "auto" | "id" |
"indentalign"}?,
  attribute indentalignlast {"left" | "center" | "right" | "auto" | "id" |
"indentalign"}?,
  attribute indentshift {length-percentage}?,
  attribute indentshiftfirst {length-percentage | "indentshift"}?,
  attribute indentshiftlast {length-percentage | "indentshift"}?,
  attribute indenttarget {idref}?,
  attribute largeop {mathml-boolean}?,
  attribute leftoverhang {length-percentage}?,
  attribute length {unsigned-integer}?,
  attribute linebreak {"auto" | "newline" | "nobreak" | "goodbreak" | "badbreak"}?,
  attribute linebreakmultchar {text}?,
  attribute linebreakstyle {"before" | "after" | "duplicate" | "infixlinebreakstyle"}?,
  attribute lineleading {length-percentage}?,
  attribute linethickness {length-percentage | "thin" | "medium" | "thick"}?,
  attribute location {"w" | "nw" | "n" | "ne" | "e" | "se" | "s" | "sw"}?,
  attribute longdivstyle {"lefttop" | "stackedrightright" | "mediumstackedrightright" |
"shortstackedrightright" | "righttop" | "left/\right" | "left)(right" | ":right=right" |
"stackedleftleft" | "stackedleftlinetop"}?,
  attribute lquote {text}?,
  attribute lspace {length-percentage}?,
  attribute mathsize {"small" | "normal" | "big" | length-percentage}?,
  attribute mathvariant {"normal" | "bold" | "italic" | "bold-italic" | "double-struck" |
"bold-fraktur" | "script" | "bold-script" | "fraktur" | "sans-serif" | "bold-sans-serif"
| "sans-serif-italic" | "sans-serif-bold-italic" | "monospace" | "initial" | "tailed" |
"looped" | "stretched"}?,
  attribute minlabelspacing {length-percentage}?,
  attribute minsize {length-percentage}?,
  attribute movablelimits {mathml-boolean}?,
  attribute mslinethickness {length-percentage | "thin" | "medium" | "thick"}?,
  attribute notation {text}?,
  attribute numalign {"left" | "center" | "right"}?,
  attribute open {text}?,
  attribute position {integer}?,
  attribute rightoverhang {length-percentage}?,
  attribute rowalign {list {verticalalign+} }?,
  attribute rowlines {list {linestyle +}}?,
  attribute rowspacing {list {(length-percentage) +}}?,
  attribute rowspan {positive-integer}?,
  attribute rquote {text}?,
  attribute rspace {length-percentage}?,
  attribute selection {positive-integer}?,
  attribute separators {text}?,
  attribute shift {integer}?,
  attribute side {"left" | "right" | "leftoverlap" | "rightoverlap"}?,
  attribute stackalign {"left" | "center" | "right" | "decimalpoint"}?,
  attribute stretchy {mathml-boolean}?,
  attribute subscriptshift {length-percentage}?,
```

```
  attribute superscriptshift {length-percentage}?,
  attribute symmetric {mathml-boolean}?,
  attribute valign {length-percentage}?,
  attribute width {length-percentage}?


math.attributes &= mstyle.specificattributes
math.attributes &= mstyle.generalattributes
math.attributes &= attribute overflow {"linebreak" | "scroll" | "elide" | "truncate" |
"scale"}?

mfenced = element mfenced {mfenced.attributes, ImpliedMrow}
mfenced.attributes =
  MathMLPGlobalAttributes,
  attribute open {text}?,
  attribute close {text}?,
  attribute separators {text}?


menclose = element menclose {menclose.attributes, ImpliedMrow}
menclose.attributes =
  MathMLPGlobalAttributes,
  attribute notation {text}?


munder.attributes &=
  attribute align {"left" | "right" | "center"}?

mover.attributes &=
  attribute align {"left" | "right" | "center"}?

munderover.attributes &=
  attribute align {"left" | "right" | "center"}?

msub.attributes &=
  attribute subscriptshift {length-percentage}?

msup.attributes &=
  attribute superscriptshift {length-percentage}?

msubsup.attributes &=
  attribute subscriptshift {length-percentage}?,
  attribute superscriptshift {length-percentage}?


mtable.attributes &=
  attribute align {xsd:string {
    pattern ='\s*(top|bottom|center|baseline|axis)(\s+-?[0-9]+)?\s*'}}?,
  attribute rowalign {list {verticalalign+} }?,
  attribute columnalign {list {columnalignstyle+} }?,
  attribute columnwidth {list {("auto" | length-percentage | "fit") +}}?,
  attribute width {"auto" | length-percentage}?,
  attribute rowspacing {list {(length-percentage) +}}?,
```

```
  attribute columnspacing {list {(length-percentage) +}}?,
  attribute rowlines {list {linestyle +}}?,
  attribute columnlines {list {linestyle +}}?,
  attribute frame {linestyle}?,
  attribute framespacing {list {length-percentage, length-percentage}}?,
  attribute equalrows {mathml-boolean}?,
  attribute equalcolumns {mathml-boolean}?,
  attribute displaystyle {mathml-boolean}?


mtr.attributes &=
  attribute rowalign {"top" | "bottom" | "center" | "baseline" | "axis"}?,
  attribute columnalign {list {columnalignstyle+} }?


mtd.attributes &=
  attribute rowalign {"top" | "bottom" | "center" | "baseline" | "axis"}?,
  attribute columnalign {columnalignstyle}?


mstack = element mstack {mstack.attributes, MstackExpression*}
mstack.attributes =
  MathMLPGlobalAttributes,
  attribute align {xsd:string {
    pattern ='\s*(top|bottom|center|baseline|axis)(\s+-?[0-9]+)?\s*'}}?,
  attribute stackalign {"left" | "center" | "right" | "decimalpoint"}?,
  attribute charalign {"left" | "center" | "right"}?,
  attribute charspacing {length-percentage | "loose" | "medium" | "tight"}?


mlongdiv = element mlongdiv {mlongdiv.attributes,
MstackExpression,MstackExpression,MstackExpression+}
mlongdiv.attributes =
  msgroup.attributes,
  attribute longdivstyle {"lefttop" | "stackedrightright" | "mediumstackedrightright" |
"shortstackedrightright" | "righttop" | "left/\right" | "left)(right" | ":right=right" |
"stackedleftleft" | "stackedleftlinetop"}?


msgroup = element msgroup {msgroup.attributes, MstackExpression*}
msgroup.attributes =
  MathMLPGlobalAttributes,
  attribute position {integer}?,
  attribute shift {integer}?


msrow = element msrow {msrow.attributes, MsrowExpression*}
msrow.attributes =
  MathMLPGlobalAttributes,
  attribute position {integer}?


mscarries = element mscarries {mscarries.attributes, (MsrowExpression|mscarry)*}
```

```
mscarries.attributes =
  MathMLPGlobalAttributes,
  attribute position {integer}?,
  attribute location {"w" | "nw" | "n" | "ne" | "e" | "se" | "s" | "sw"}?,
  attribute crossout {list {("none" | "updiagonalstrike" | "downdiagonalstrike" |
"verticalstrike" | "horizontalstrike")*}}?,
  attribute scriptsizemultiplier {number}?


mscarry = element mscarry {mscarry.attributes, MsrowExpression*}
mscarry.attributes =
  MathMLPGlobalAttributes,
  attribute location {"w" | "nw" | "n" | "ne" | "e" | "se" | "s" | "sw"}?,
  attribute crossout {list {("none" | "updiagonalstrike" | "downdiagonalstrike" |
"verticalstrike" | "horizontalstrike")*}}?
```

A.2.3 Strict Content MathML

The grammar for Strict Content MathML 4 can be found at https://www.w3.org/Math/RelaxNG/mathml4/mathml4-strict-content.rnc.

```
# MathML 4 (Strict Content)
# ########################

#     Copyright 1998-2024 W3C (MIT, ERCIM, Keio, Beihang)
#
#     Use and distribution of this code are permitted under the terms
#     W3C Software Notice and License
#     http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231


default namespace m = "http://www.w3.org/1998/Math/MathML"

start |= math.strict

CommonAtt =
    attribute id {xsd:ID}?,
    attribute xref {text}?

math.strict = element math {math.attributes,ContExp*}

math.attributes &= CommonAtt

ContExp = semantics-contexp | cn | ci | csymbol | apply | bind | share | cerror | cbytes
| cs

cn = element cn {cn.attributes,cn.content}
cn.content = text
cn.attributes = CommonAtt, attribute type {"integer" | "real" | "double" | "hexdouble"}
```

```
semantics-ci = element semantics {CommonAtt,(ci|semantics-ci),
  (annotation|annotation-xml)*}

semantics-contexp = element semantics {CommonAtt,MathExpression,
  (annotation|annotation-xml)*}

annotation |= element annotation {CommonAtt,text}

anyElement |=  element (* - m:*) {(attribute * {text}|text| anyElement)*}

annotation-xml |= element annotation-xml {annotation-xml.attributes,
                                          (MathExpression*|anyElement*)}

annotation-xml.attributes &= CommonAtt, cd?, encoding?

encoding &= attribute encoding {xsd:string}




ci = element ci {ci.attributes, ci.content}
ci.attributes = CommonAtt, ci.type?
ci.type = attribute type {"integer" | "rational" | "real" | "complex" | "complex-polar" |
"complex-cartesian" | "constant" | "function" | "vector" | "list" | "set" | "matrix"}
ci.content = text


csymbol = element csymbol {csymbol.attributes,csymbol.content}

SymbolName = xsd:NCName
csymbol.attributes = CommonAtt, cd
csymbol.content = SymbolName
cd = attribute cd {xsd:NCName}
name = attribute name {xsd:NCName}
src = attribute src {xsd:anyURI}?

BvarQ = bvar*
bvar = element bvar {CommonAtt, (ci | semantics-ci)}

apply = element apply {CommonAtt,apply.content}
apply.content = ContExp+


bind = element bind {CommonAtt,bind.content}
bind.content = ContExp,bvar*,ContExp

share = element share {CommonAtt, src, empty}

cerror = element cerror {cerror.attributes, csymbol, ContExp*}
cerror.attributes = CommonAtt

cbytes = element cbytes {cbytes.attributes, base64}
cbytes.attributes = CommonAtt
```

```
base64 = xsd:base64Binary

cs = element cs {cs.attributes, text}
cs.attributes = CommonAtt

MathExpression |= ContExp
```

A.2.4 Content MathML

The grammar for Content MathML 4 builds on the grammar for the Strict Content MathML subset, and can be found at https://www.w3.org/Math/RelaxNG/mathml4/mathml4-content.rnc.

```
# MathML 4 (Content)
# ##################

#     Copyright 1998-2024 W3C (MIT, ERCIM, Keio, Beihang)
#
#     Use and distribution of this code are permitted under the terms
#     W3C Software Notice and License
#     http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

default namespace m = "http://www.w3.org/1998/Math/MathML"
namespace local = ""

include "mathml4-strict-content.rnc"{
  cn.content = (text | sep | PresentationExpression)*
  cn.attributes = CommonAtt, DefEncAtt, attribute type {text}?, base?

  ci.attributes = CommonAtt, DefEncAtt, ci.type?
  ci.type = attribute type {text}
  ci.content = (text | PresentationExpression)*

  csymbol.attributes = CommonAtt, DefEncAtt, attribute type {text}?,cd?
  csymbol.content = (text | PresentationExpression)*

  annotation-xml.attributes |= CommonAtt, cd?, name?, encoding?

  bvar = element bvar {CommonAtt, ((ci | semantics-ci) & degree?)}

  cbytes.attributes = CommonAtt, DefEncAtt

  cs.attributes = CommonAtt, DefEncAtt

  apply.content = ContExp+ | (ContExp, BvarQ, Qualifier*, ContExp*)

  bind.content = apply.content
}

NonMathMLAtt |= attribute (* - (local:*|m:*)) {xsd:string}
```

```
math.attributes &=
    attribute alttext {text}?

MathMLDataAttributes &=
  attribute data-other {text}?

CommonAtt &=
                   NonMathMLAtt*,
                   MathMLDataAttributes,
                   attribute class {xsd:NCName}?,
                   attribute style {xsd:string}?,
                   attribute href {xsd:anyURI}?,
                   attribute intent {text}?,
                   attribute arg {xsd:NCName}?

base = attribute base {text}


sep = element sep {empty}
PresentationExpression |= notAllowed
DefEncAtt = attribute encoding {xsd:string}?,
            attribute definitionURL {xsd:anyURI}?


DomainQ = (domainofapplication|condition|interval|(lowlimit,uplimit?))*
domainofapplication = element domainofapplication {ContExp}
condition = element condition {ContExp}
uplimit = element uplimit {ContExp}
lowlimit = element lowlimit {ContExp}

Qualifier = DomainQ|degree|momentabout|logbase
degree = element degree {ContExp}
momentabout = element momentabout {ContExp}
logbase = element logbase {ContExp}

type = attribute type {text}
order = attribute order {"numeric" | "lexicographic"}
closure = attribute closure {text}


ContExp |= piecewise


piecewise = element piecewise {CommonAtt, DefEncAtt,(piece* & otherwise?)}

piece = element piece {CommonAtt, DefEncAtt, ContExp, ContExp}

otherwise = element otherwise {CommonAtt, DefEncAtt, ContExp}


interval.class = interval
ContExp |= interval.class
```

```
interval = element interval { CommonAtt, DefEncAtt,closure?, ContExp,ContExp}

unary-functional.class = inverse | ident | domain | codomain | image | ln | log | moment
ContExp |= unary-functional.class


inverse = element inverse { CommonAtt, DefEncAtt, empty}
ident = element ident { CommonAtt, DefEncAtt, empty}
domain = element domain { CommonAtt, DefEncAtt, empty}
codomain = element codomain { CommonAtt, DefEncAtt, empty}
image = element image { CommonAtt, DefEncAtt, empty}
ln = element ln { CommonAtt, DefEncAtt, empty}
log = element log { CommonAtt, DefEncAtt, empty}
moment = element moment { CommonAtt, DefEncAtt, empty}

lambda.class = lambda
ContExp |= lambda.class


lambda = element lambda { CommonAtt, DefEncAtt, BvarQ, DomainQ, ContExp}

nary-functional.class = compose
ContExp |= nary-functional.class


compose = element compose { CommonAtt, DefEncAtt, empty}

binary-arith.class = quotient | divide | minus | power | rem | root
ContExp |= binary-arith.class


quotient = element quotient { CommonAtt, DefEncAtt, empty}
divide = element divide { CommonAtt, DefEncAtt, empty}
minus = element minus { CommonAtt, DefEncAtt, empty}
power = element power { CommonAtt, DefEncAtt, empty}
rem = element rem { CommonAtt, DefEncAtt, empty}
root = element root { CommonAtt, DefEncAtt, empty}

unary-arith.class = factorial | minus | root | abs | conjugate | arg | real | imaginary |
floor | ceiling | exp
ContExp |= unary-arith.class


factorial = element factorial { CommonAtt, DefEncAtt, empty}
abs = element abs { CommonAtt, DefEncAtt, empty}
conjugate = element conjugate { CommonAtt, DefEncAtt, empty}
arg = element arg { CommonAtt, DefEncAtt, empty}
real = element real { CommonAtt, DefEncAtt, empty}
imaginary = element imaginary { CommonAtt, DefEncAtt, empty}
floor = element floor { CommonAtt, DefEncAtt, empty}
ceiling = element ceiling { CommonAtt, DefEncAtt, empty}
exp = element exp { CommonAtt, DefEncAtt, empty}
```

```
nary-minmax.class = max | min
ContExp |= nary-minmax.class


max = element max { CommonAtt, DefEncAtt, empty}
min = element min { CommonAtt, DefEncAtt, empty}

nary-arith.class = plus | times | gcd | lcm
ContExp |= nary-arith.class


plus = element plus { CommonAtt, DefEncAtt, empty}
times = element times { CommonAtt, DefEncAtt, empty}
gcd = element gcd { CommonAtt, DefEncAtt, empty}
lcm = element lcm { CommonAtt, DefEncAtt, empty}

nary-logical.class = and | or | xor
ContExp |= nary-logical.class


and = element and { CommonAtt, DefEncAtt, empty}
or = element or { CommonAtt, DefEncAtt, empty}
xor = element xor { CommonAtt, DefEncAtt, empty}

unary-logical.class = not
ContExp |= unary-logical.class


not = element not { CommonAtt, DefEncAtt, empty}

binary-logical.class = implies | equivalent
ContExp |= binary-logical.class


implies = element implies { CommonAtt, DefEncAtt, empty}
equivalent = element equivalent { CommonAtt, DefEncAtt, empty}

quantifier.class = forall | exists
ContExp |= quantifier.class


forall = element forall { CommonAtt, DefEncAtt, empty}
exists = element exists { CommonAtt, DefEncAtt, empty}

nary-reln.class = eq | gt | lt | geq | leq
ContExp |= nary-reln.class


eq = element eq { CommonAtt, DefEncAtt, empty}
gt = element gt { CommonAtt, DefEncAtt, empty}
lt = element lt { CommonAtt, DefEncAtt, empty}
geq = element geq { CommonAtt, DefEncAtt, empty}
```

```
leq = element leq { CommonAtt, DefEncAtt, empty}

binary-reln.class = neq | approx | factorof | tendsto
ContExp |= binary-reln.class


neq = element neq { CommonAtt, DefEncAtt, empty}
approx = element approx { CommonAtt, DefEncAtt, empty}
factorof = element factorof { CommonAtt, DefEncAtt, empty}
tendsto = element tendsto { CommonAtt, DefEncAtt, type?, empty}

int.class = int
ContExp |= int.class


int = element int { CommonAtt, DefEncAtt, empty}

Differential-Operator.class = diff
ContExp |= Differential-Operator.class


diff = element diff { CommonAtt, DefEncAtt, empty}

partialdiff.class = partialdiff
ContExp |= partialdiff.class


partialdiff = element partialdiff { CommonAtt, DefEncAtt, empty}

unary-veccalc.class = divergence | grad | curl | laplacian
ContExp |= unary-veccalc.class


divergence = element divergence { CommonAtt, DefEncAtt, empty}
grad = element grad { CommonAtt, DefEncAtt, empty}
curl = element curl { CommonAtt, DefEncAtt, empty}
laplacian = element laplacian { CommonAtt, DefEncAtt, empty}

nary-setlist-constructor.class = set | \list
ContExp |= nary-setlist-constructor.class


set = element set { CommonAtt, DefEncAtt, type?, BvarQ*, DomainQ*, ContExp*}
\list = element \list { CommonAtt, DefEncAtt, order?, BvarQ*, DomainQ*, ContExp*}

nary-set.class = union | intersect | cartesianproduct
ContExp |= nary-set.class


union = element union { CommonAtt, DefEncAtt, empty}
intersect = element intersect { CommonAtt, DefEncAtt, empty}
cartesianproduct = element cartesianproduct { CommonAtt, DefEncAtt, empty}
```

```
binary-set.class = in | notin | notsubset | notprsubset | setdiff
ContExp |= binary-set.class


in = element in { CommonAtt, DefEncAtt, empty}
notin = element notin { CommonAtt, DefEncAtt, empty}
notsubset = element notsubset { CommonAtt, DefEncAtt, empty}
notprsubset = element notprsubset { CommonAtt, DefEncAtt, empty}
setdiff = element setdiff { CommonAtt, DefEncAtt, empty}

nary-set-reln.class = subset | prsubset
ContExp |= nary-set-reln.class


subset = element subset { CommonAtt, DefEncAtt, empty}
prsubset = element prsubset { CommonAtt, DefEncAtt, empty}

unary-set.class = card
ContExp |= unary-set.class


card = element card { CommonAtt, DefEncAtt, empty}

sum.class = sum
ContExp |= sum.class


sum = element sum { CommonAtt, DefEncAtt, empty}

product.class = product
ContExp |= product.class


product = element product { CommonAtt, DefEncAtt, empty}

limit.class = limit
ContExp |= limit.class


limit = element limit { CommonAtt, DefEncAtt, empty}

unary-elementary.class = sin | cos | tan | sec | csc | cot | sinh | cosh | tanh | sech |
csch | coth | arcsin | arccos | arctan | arccosh | arccot | arccoth | arccsc | arccsch |
arcsec | arcsech | arcsinh | arctanh
ContExp |= unary-elementary.class


sin = element sin { CommonAtt, DefEncAtt, empty}
cos = element cos { CommonAtt, DefEncAtt, empty}
tan = element tan { CommonAtt, DefEncAtt, empty}
sec = element sec { CommonAtt, DefEncAtt, empty}
csc = element csc { CommonAtt, DefEncAtt, empty}
cot = element cot { CommonAtt, DefEncAtt, empty}
```

```
sinh = element sinh { CommonAtt, DefEncAtt, empty}
cosh = element cosh { CommonAtt, DefEncAtt, empty}
tanh = element tanh { CommonAtt, DefEncAtt, empty}
sech = element sech { CommonAtt, DefEncAtt, empty}
csch = element csch { CommonAtt, DefEncAtt, empty}
coth = element coth { CommonAtt, DefEncAtt, empty}
arcsin = element arcsin { CommonAtt, DefEncAtt, empty}
arccos = element arccos { CommonAtt, DefEncAtt, empty}
arctan = element arctan { CommonAtt, DefEncAtt, empty}
arccosh = element arccosh { CommonAtt, DefEncAtt, empty}
arccot = element arccot { CommonAtt, DefEncAtt, empty}
arccoth = element arccoth { CommonAtt, DefEncAtt, empty}
arccsc = element arccsc { CommonAtt, DefEncAtt, empty}
arccsch = element arccsch { CommonAtt, DefEncAtt, empty}
arcsec = element arcsec { CommonAtt, DefEncAtt, empty}
arcsech = element arcsech { CommonAtt, DefEncAtt, empty}
arcsinh = element arcsinh { CommonAtt, DefEncAtt, empty}
arctanh = element arctanh { CommonAtt, DefEncAtt, empty}

nary-stats.class = mean | median | mode | sdev | variance
ContExp |= nary-stats.class


mean = element mean { CommonAtt, DefEncAtt, empty}
median = element median { CommonAtt, DefEncAtt, empty}
mode = element mode { CommonAtt, DefEncAtt, empty}
sdev = element sdev { CommonAtt, DefEncAtt, empty}
variance = element variance { CommonAtt, DefEncAtt, empty}

nary-constructor.class = vector | matrix | matrixrow
ContExp |= nary-constructor.class


vector = element vector { CommonAtt, DefEncAtt, BvarQ, DomainQ, ContExp*}
matrix = element matrix { CommonAtt, DefEncAtt, BvarQ, DomainQ, ContExp*}
matrixrow = element matrixrow { CommonAtt, DefEncAtt, BvarQ, DomainQ, ContExp*}

unary-linalg.class = determinant | transpose
ContExp |= unary-linalg.class


determinant = element determinant { CommonAtt, DefEncAtt, empty}
transpose = element transpose { CommonAtt, DefEncAtt, empty}

nary-linalg.class = selector
ContExp |= nary-linalg.class


selector = element selector { CommonAtt, DefEncAtt, empty}

binary-linalg.class = vectorproduct | scalarproduct | outerproduct
ContExp |= binary-linalg.class
```

```
vectorproduct = element vectorproduct { CommonAtt, DefEncAtt, empty}
scalarproduct = element scalarproduct { CommonAtt, DefEncAtt, empty}
outerproduct = element outerproduct { CommonAtt, DefEncAtt, empty}

constant-set.class = integers | reals | rationals | naturalnumbers | complexes | primes |
emptyset
ContExp |= constant-set.class


integers = element integers { CommonAtt, DefEncAtt, empty}
reals = element reals { CommonAtt, DefEncAtt, empty}
rationals = element rationals { CommonAtt, DefEncAtt, empty}
naturalnumbers = element naturalnumbers { CommonAtt, DefEncAtt, empty}
complexes = element complexes { CommonAtt, DefEncAtt, empty}
primes = element primes { CommonAtt, DefEncAtt, empty}
emptyset = element emptyset { CommonAtt, DefEncAtt, empty}

constant-arith.class = exponentiale | imaginaryi | notanumber | true | false | pi |
eulergamma | infinity
ContExp |= constant-arith.class


exponentiale = element exponentiale { CommonAtt, DefEncAtt, empty}
imaginaryi = element imaginaryi { CommonAtt, DefEncAtt, empty}
notanumber = element notanumber { CommonAtt, DefEncAtt, empty}
true = element true { CommonAtt, DefEncAtt, empty}
false = element false { CommonAtt, DefEncAtt, empty}
pi = element pi { CommonAtt, DefEncAtt, empty}
eulergamma = element eulergamma { CommonAtt, DefEncAtt, empty}
infinity = element infinity { CommonAtt, DefEncAtt, empty}
```

A.2.5 Full MathML

The grammar for full MathML 4 is simply a merger of the above grammars, and can be found at https://www.w3.org/Math/RelaxNG/mathml4/mathml4.rnc.

```
# MathML 4 (full)
# ###############

#     Copyright 1998-2024 W3C (MIT, ERCIM, Keio)
#
#     Use and distribution of this code are permitted under the terms
#     W3C Software Notice and License
#     http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

default namespace m = "http://www.w3.org/1998/Math/MathML"

# Presentation MathML
include "mathml4-presentation.rnc"  {
```

```
anyElement =  element (* - m:*) {(attribute * {text}|text| anyElement)*}
}


# Content  MathML
include "mathml4-content.rnc"
```

A.2.6 Legacy MathML

Some elements and attributes that were deprecated in MathML 3 are removed from MathML 4. This schema extends the full MathML 4 schema, adding these constructs back, allowing validation of existing MathML documents. It can be found at https://www.w3.org/Math/RelaxNG/mathml4/mathml4-legacy.rnc.

```
# MathML 4 (legacy)
# ################

#     Copyright 1998-2024 W3C (MIT, ERCIM, Keio)
#
#     Use and distribution of this code are permitted under the terms
#     W3C Software Notice and License
#     http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

default namespace m = "http://www.w3.org/1998/Math/MathML"

# MathML 4
include "mathml4.rnc" {

# unitless lengths
length-percentage = xsd:string {
  pattern = '\s*((-?[0-9]*([0-9]\.?|\.[0-9])[0-9]*(e[mx]|in|cm|mm|p[xtc]|%)?)|(negative)?
((very){0,2}thi(n|ck)|medium)mathspace)\s*'
}
}

# Removed MathML 1/2/3 elements

ContExp |= reln | fn | declare

reln = element reln {ContExp*}
fn = element fn {ContExp}
declare = element declare {attribute type {xsd:string}?,
                           attribute scope {xsd:string}?,
                           attribute nargs {xsd:nonNegativeInteger}?,
                           attribute occurrence {"prefix"|"infix"|"function-model"}?,
                           DefEncAtt,
                           ContExp+}



# legacy attributes
```

```
CommonAtt &= attribute other {text}?
MathMLPGlobalAttributes &= attribute other {text}?


mglyph.deprecatedattributes =
  attribute fontfamily {text}?,
  attribute index {integer}?,
  attribute mathvariant {"normal" | "bold" | "italic" | "bold-italic" | "double-struck" |
"bold-fraktur" | "script" | "bold-script" | "fraktur" | "sans-serif" | "bold-sans-serif"
| "sans-serif-italic" | "sans-serif-bold-italic" | "monospace" | "initial" | "tailed" |
"looped" | "stretched"}?,
  attribute mathsize {"small" | "normal" | "big" | length-percentage}?

mglyph.attributes &= mglyph.deprecatedattributes

mstyle.deprecatedattributes =
  attribute veryverythinmathspace {length-percentage}?,
  attribute verythinmathspace {length-percentage}?,
  attribute thinmathspace {length-percentage}?,
  attribute mediummathspace {length-percentage}?,
  attribute thickmathspace {length-percentage}?,
  attribute verythickmathspace {length-percentage}?,
  attribute veryverythickmathspace {length-percentage}?

mstyle.attributes &= mstyle.deprecatedattributes


math.deprecatedattributes = attribute mode {xsd:string}?,
                            attribute macros {xsd:string}?

math.attributes &= math.deprecatedattributes


DeprecatedTokenAtt =
  attribute fontfamily {text}?,
  attribute fontweight {"normal" | "bold"}?,
  attribute fontstyle {"normal" | "italic"}?,
  attribute fontsize {length-percentage}?,
  attribute color {color}?,
  attribute background {color}?,
  attribute mathsize {"small" | "normal" | "big" }?

DeprecatedMoAtt =
  attribute fence {mathml-boolean}?,
  attribute separator {mathml-boolean}?

mstyle.attributes &= DeprecatedTokenAtt
mstyle.attributes &= DeprecatedMoAtt
mglyph.attributes &= DeprecatedTokenAtt
mn.attributes &= DeprecatedTokenAtt
mi.attributes &= DeprecatedTokenAtt
mo.attributes &= DeprecatedTokenAtt
```

```
mo.attributes &= DeprecatedMoAtt
mtext.attributes &= DeprecatedTokenAtt
mspace.attributes &= DeprecatedTokenAtt
ms.attributes &= DeprecatedTokenAtt

semantics.attributes &= DefEncAtt


# malignmark in tokens
token.content |= malignmark
# malignmark in mfrac etc
MathExpression |= MalignExpression

maligngroup.attributes &=
  attribute groupalign {"left" | "center" | "right" | "decimalpoint"}?

malignmark.attributes &=
  attribute edge {"left" | "right"}?

mstyle.generalattributes &=
  attribute edge {"left" | "right"}?

# groupalign
group-alignment = "left" | "center" | "right" | "decimalpoint"
group-alignment-list = list {group-alignment+}
group-alignment-list-list = xsd:string {
  pattern = '(\s*\{\s*(left|center|right|decimalpoint)(\s+(left|center|right|
decimalpoint))*\})*\s*' }

mstyle.generalattributes &=
  attribute groupalign {group-alignment-list-list}?

mtable.attributes &=
  attribute groupalign {group-alignment-list-list}?,
  attribute alignmentscope {list {("true" | "false") +}}?,
  attribute side {"left" | "right" | "leftoverlap" | "rightoverlap"}?,
  attribute minlabelspacing {length-percentage}?

mtr.attributes &=
  attribute groupalign {group-alignment-list-list}?

mtd.attributes &=
  attribute groupalign {group-alignment-list}?

mlabeledtr = element mlabeledtr {mlabeledtr.attributes, mtd+}
mlabeledtr.attributes =
  mtr.attributes

TableRowExpression |= mlabeledtr
```

A.3 Using the MathML DTD

The MathML DTD uses the strategy outlined in [Modularization] to allow the use of namespace prefixes on MathML elements. However it is recommended that namespace prefixes are not used in XML serialization of MathML, for compatibility with the HTML serialization.

Note that unlike the HTML serialization, when using the XML serialization, character entity references such as `&int;` may not be used unless a DTD is specified, either the full MathML DTD as described here or the set of HTML/MathML entity declarations as specified by [Entities]. Characters may always be specified by using character data directly, or numeric character references, so ∫ or `&#x222B;` rather than `&int;`.

A.4 Using the MathML XML Schema

MathML fragments can be validated using the XML Schema for MathML, located at http://www.w3.org/Math/XMLSchema/mathml4/mathml4.xsd. The provided schema has been mechanically generated from the Relax NG schema, it omits some constraints that can not be enforced using XSD syntax.

B. Operator Dictionary

The following table gives the suggested dictionary of rendering properties for operators, fences, separators, and accents in MathML, all of which are represented by `mo` elements. For brevity, all such elements will be called simply “operators” in this Appendix. Note that implementations of [MathML-Core] will use these values as normative definitions of the default operator spacing.

B.1 Indexing of the operator dictionary

The dictionary is indexed not just by the element content, but by the element content and `form` attribute value, together. Operators with more than one possible form have more than one entry. The MathML specification specifies which form to use when no `form` attribute is given; see 3.2.5.6.2 Default value of the `form` attribute.

The data is all available in machine readable form in `unicode.xml` which is also the source of the HTML/MathML entity definitions and distributed with [Entities]. It is however presented below in two more human readable formats. See also [MathML-Core] for an alternative presentation of the data that is used in that specification.

In the first presentation, operators are ordered first by the form and spacing attributes, and then by `priority`. The characters are then listed, with additional data on remaining operator dictionary entries for that character given via a title attribute which will appear as a popup tooltip in suitable browsers.

In the second presentation of the data, the rows of the table may be reordered in suitable browsers by clicking on a heading in the top row, to cause the table to be ordered by that column.

B.2 Notes on `lspace` and `rspace` attributes

The values for `lspace` and `rspace` given here range from 0 to 7 denoting multiples of 1/18 em matching the values used for namedspace.

For the invisible operators whose content is `InvisibleTimes` or `ApplyFunction`, it is suggested that MathML renderers choose spacing in a context-sensitive way (which is an exception to the static values given in the following table). For `<mo>&ApplyFunction;</mo>`, the total spacing (`lspace`+`rspace`) in expressions such as sin x (where the right operand doesn't start with a fence) should be greater than zero; for `<mo>&InvisibleTimes;</mo>`, the total spacing should be greater than zero when both operands (or the nearest tokens on either side, if on the baseline) are identifiers displayed in a non-slanted font (i.e.. under the suggested rules, when both operands are multi-character identifiers).

Some renderers may wish to use no spacing for most operators appearing in scripts (i.e. when `scriptlevel` is greater than 0; see 3.3.4 Style Change `<mstyle>`), as is the case in TeX.

B.3 Operator dictionary entries

B.3.1 Compressed view

▸ Show Section

form:infix lspace:0 rspace:0

Priority: 160

⟨*invisible separator*⟩

Priority: 620

⟨*invisible times*⟩

Priority: 660

\

Priority: 720

Δ

Priority: 880

⟨*function application*⟩

Priority: 920

⟨*invisible plus*⟩

Priority: 940

–

form:infix lspace:0 rspace:3

Priority: 140

;

Priority: 160

,

Priority: 180

:

form:infix lspace:3 rspace:3

Priority: 560

@

Priority: 620

*, ., ·, ×, •, ⁃, ∗, ∙, ≀, ⊗, ⊙, ⊛, ⊠, ⊡, ⊺, ⋅, ⋆, ⋇, ⋉, ⋊, ⋋, ⋌, ⊼, ⩞, ⧆, ⧈, ⧑, ⧒, ⧖, ⧗, ⧢, ⨝, ⨞, ⨯, ⨰, ⨱, ⨲, ⨳, ⨴, ⨵, ⨶, ⨷, ⨻, ⨼, ⨽, ⨿, ⩐

Priority: 640

%

Priority: 680

⟋, ⟍, ▯

Priority: 700

⧏, ⧐

Priority: 740

⫝̸, ⫝

Priority: 760

**

Priority: 800

<>, ^

Priority: 840

?

Priority: 900

∘, ⊚, ⋄, ⧇

form:infix lspace:4 rspace:4

Priority: 360

∪, ⊌, ⊍, ⊎, ⊔, ⋓, ⩁, ⩂, ⩅, ⩊, ⩌, ⩏

Priority: 380

∩, ⊓, ⋒, ⨟, ⨠, ⨡, ⨾, ⩀, ⩃, ⩄, ⩆, ⩇, ⩈, ⩉, ⩋, ⩍, ⩎, ⫛

Priority: 400

+, -, ±, −, ∓, ∔, ∖, ∨, ∸, ⊕, ⊖, ⊝, ⊞, ⊟, ⊽, ⋎, ➕, ➖, ⦸, ⧅, ⧵, ⧷, ⧹, ⧺, ⧻, ⨢, ⨣, ⨤, ⨥, ⨦, ⨧, ⨨, ⨩, ⨪, ⨫, ⨬, ⨭, ⨮, ⨹, ⨺, ⩒, ⩔, ⩖, ⩗, ⩛, ⩝, ⩡, ⩢, ⩣

Priority: 420

⊻

Priority: 600

&&, ∧, ⊼, ⋏, ⩑, ⩓, ⩕, ⩘, ⩙, ⩚, ⩜, ⩞, ⩟, ⩠

Priority: 680

/, ÷, ⁄, ∕, ∶, ⊘, ➗, ⦼, ⧄, ⧶, ⧸, ⨸, ⫶, ⫻, ⫽

form:infix lspace:5 rspace:5

Priority: 140

⦁

Priority: 180

⦂

Priority: 220

->, ⊶, ⊷, ⊸, ⧴

Priority: 240

//

Priority: 260

⊢, ⊣, ⊦, ⊨, ⊩, ⊪, ⊫, ⊬, ⊭, ⊮, ⊯, ⊣, ⊤, ⊥, ⫫, ⫤, ⫣, ⫪, ⫧, ⫨, ⫩, ⫟, ⫠, ⫡, ⫢

Priority: 300

∈, ∉, ∊, ∋, ∌, ∍, ⊂, ⊃, ⊄, ⊅, ⊆, ⊇, ⊈, ⊉, ⊊, ⊋, ⊏, ⊐, ⊑, ⊒, ⊰, ⊱, ◁, ▷, ⋲, ⋳, ⋪, ⋫, ⋬, ⋭, ∈, ⋵, ⋹, ⋸, ⋶, ⋷, ⋺, ⋻, ⋽, ⋾, ⋼, ⋿, ⊏̸, ⊐̸, ⊲, ⊳, ⊴, ⊵, ⫋, ⫌, ⫅, ⫆, ⫇, ⫈, ⫉, ⫊, ⫍, ⫎, ⫏, ⫐, ⫑, ⫒, ⫓, ⫔, ⫕, ⫖, ⫗, ⫘, ⫙, ⫷, ⫸, ⫹, ⫺

Priority: 320

!=, *=, +=, -=, /=, :=, <, <=, =, ==, >, >=, |, ||, ∝, ∣, ∤, ∥, ∦, ∷, -:, ∺, ∻, ∼, ∽, ∾, ≁, ≂, ≃, ≄, ≅, ≆, ≇, ≈, ≉, ≊, ≋, ≌, ≍, ≎, ≏, ≐, ≑, ≒, ≓, ≔, ≕, ≖, ≗, ≙, ≚, ≛, ≜, ≝, ≞, ≟, ≠, ≡, ≢, ≣, ≤, ≥, ≦, ≧, ≨, ≩, ≪, ≫, ≬, ≺, ≻, ≼, ≽, ≾, ≿, ⊀, ⊁, ⊜, ⊧, ⊿, ⊾, ⋈, ⋌, ⋔, ⋕, ⋖, ⋗, ⋘, ⋙, ⋚, ⋛, ⋜, ⋝, ⋞, ⋟, ⋠, ⋡, ⋢, ⋣, ⋤, ⋥, ⋦, ⋧, ⋨, ⋩, ⋮, ⋯, ⋰, ⋱, ⟀, ⦁, ⦂, ⦙, ⦚, ⦛, ⦜, ⦝, ⧀, ⧁, ⧣, ⧤, ⧥, ⩦, ⩧, ⩪, ⩫, ⩬, ⩭, ⩮, ⩯, ⩰, ⩱, ⩲, ⩳, ⩴, ⩵, ⩶, ⩷, ⩸, ⩹, ⩺, ⩻, ⩼, ⩽, ⩾, ⩿, ⪀, ⪁, ⪂, ⪃, ⪄, ⪅, ⪆, ⪇, ⪈, ⪉, ⪊, ⪋, ⪌, ⪍, ⪎, ⪏, ⪐, ⪑, ⪒, ⪓, ⪔, ⪕, ⪖, ⪗, ⪘, ⪙, ⪚, ⪛, ⪜, ⪝, ⪞, ⪟, ⪠, ⪡, ⪢, ⪣, ⪤, ⪥, ⪦, ⪧, ⪨, ⪩, ⪪, ⪫, ⪬, ⪭, ⪮, ⪯, ⪰, ⪱, ⪲, ⪳, ⪴, ⪵, ⪶, ⪷, ⪸, ⪹, ⪺, ⪻, ⪼, ⪽, ⪾, ⪿, ⫀, ⫁, ⫂, ⫃, ⫄, ⫝̸, ⫴, ⫵, ⫶, ⫻, ⫼, ⫽, ⫾, ⫿, 🈑

Priority: 340

←, ↑, →, ↓, ↔, ↕, ↖, ↗, ↘, ↙, ↚, ↛, ↜, ↝, ↞, ↟, ↠, ↡, ↢, ↣, ↤, ↥, ↦, ↧, ↨, ↩, ↪, ↫, ↬, ↭, ↮, ↯, ↰, ↱, ↲, ↳, ↴, ↵, ↶, ↷, ↸, ↹, ↺, ↻, ↼, ↽, ↾, ↿, ⇀, ⇁, ⇂, ⇃, ⇄, ⇅, ⇆, ⇇, ⇈, ⇉, ⇊, ⇋, ⇌, ⇍, ⇎, ⇏, ⇐, ⇑, ⇒, ⇓, ⇔, ⇕, ⇖, ⇗, ⇘, ⇙, ⇚, ⇛, ⇜, ⇝, ⇞, ⇟, ⇠, ⇡, ⇢, ⇣, ⇤, ⇥, ⇦, ⇧, ⇨, ⇩, ⇪, ⇫, ⇬, ⇭, ⇮, ⇯, ⇰, ⇱, ⇲, ⇳, ⇴, ⇵, ⇶, ⇷, ⇸, ⇹, ⇺, ⇻, ⇼, ⇽, ⇾, ⇿, ➔, →, ➔, ➙, ➚, ➛, ➜, ➝, ➞, ➟, ➠, ➡, ➢, ➣, ➪, ➫, ➬, ➭, ➮, ➯, ➱, ➲, ➳, ➵, ➸, ➻, ➼, ➽, ⟰, ⟱, ⟲, ⟳, ⟴, ⟵, ⟶, ⟷, ⟸, ⟹, ⟺, ⟻, ⟼, ⟽, ⟾, ⟿, ⤀, ⤁, ⤂, ⤃, ⤄, ⤅, ⤆, ⤇, ⤈, ⤉, ⤊, ⤋, ⤌, ⤍, ⤎, ⤏, ⤐, ⤑, ⤒, ⤓, ⤔, ⤕, ⤖, ⤗, ⤘, ⤙, ⤚, ⤛, ⤜, ⤝, ⤞, ⤟, ⤠, ⤡, ⤢, ⤣, ⤤, ⤥, ⤦, ⤧, ⤨, ⤩, ⤪, ⤫, ⤬, ⤭, ⤮, ⤯, ⤰, ⤱, ⤲, ⤳, ⤴, ⤵, ⤶, ⤷, ⤸, ⤹, ⤺, ⤻, ⤼, ⤽, ⤾, ⤿, ⥀, ⥁, ⥂, ⥃, ⥄, ⥅, ⥆, ⥇, ⥈, ⥉, ⥊, ⥋, ⥌, ⥍, ⥎, ⥏, ⥐, ⥑, ⥒, ⥓, ⥔, ⥕, ⥖, ⥗, ⥘, ⥙, ⥚, ⥛, ⥜, ⥝, ⥞, ⥟, ⥠, ⥡, ⥢, ⥣, ⥤, ⥥, ⥦, ⥧, ⥨, ⥩, ⥪, ⥫, ⥬, ⥭, ⥮, ⥯, ⥰, ⥱, ⥲, ⥳, ⥴, ⥵, ⥶, ⥷, ⥸, ⥹, ⥺, ⥻, ⥼, ⥽, ⥾, ⥿, ⦸, ⧂, ⧃, ⧄, ⧅, ⧆, ⧇, ⧈, ⧉, ⧊, ⧋, ⧌, ⧍, ⧎, ⧏, ⧐, ⧑, ⧒, ⧓, ⧔, ⧕, ⧖, ⧗, ⧘, ⧙, ⧛, ⧝, ⧞, ⧟, ⧠, ⧡, ⧢, ⧣, ⧤, ⧥, ⧦, ⧧, ⧨, ⧩, ⧪, ⧫, ⧬, ⧭, ⧮, ⧯, ⧰, ⧱, ⧲, ⧳, ⧴, ⧵, ⧶, ⧷, ⧸, ⧹, ⧺, ⧻, ⨀, ⨁, ⨂, ⨃, ⨄, ⨅, ⨆, ⨇, ⨈, ⨉, ⨊, ⨋, ⨌, ⨍, ⨎, ⨏, ⨐, ⨑, ⨒, ⨓, ⨔, ⨕, ⨖, ⨗, ⨘, ⨙, ⨚, ⨛, ⨜, ⨝, ⨞, ⨟, ⨠, ⨡, ⨢, ⨣, ⨤, ⨥, ⨦, ⨧, ⨨, ⨩, ⨪, ⨫, ⨬, ⨭, ⨮, ⨯, ⨰, ⨱, ⨲, ⨳, ⨴, ⨵, ⨶, ⨷, ⨸, ⨹, ⨺, ⨻, ⨼, ⨽, ⩀, ⩁, ⩂, ⩃, ⩄, ⩅, ⩆, ⩇, ⩈, ⩉, ⩊, ⩋, ⩌, ⩍, ⩎, ⩏, ⩐, ⩑, ⩒, ⩓, ⩔, ⩕, ⩖, ⩗, ⩘, ⩙, ⩚, ⩛, ⩜, ⩝, ⩞, ⩟, ⩠, ⩡, ⩢, ⩣, ⩤, ⩥, ⩦, ⩧, ⩨, ⩩, ⩪, ⩫, ⩬, ⩭, ⩮, ⩯, ⩰, ⩱, ⩲, ⩳, ⩴, ⩵, ⩶, ⩷, ⩸, ⩹, ⩺, ⩻, ⩼, ⩽, ⩾, ⩿, ⪀, ⪁, ⪂, ⪃, ⪄, ⪅, ⪆, ⪇, ⪈, ⪉, ⪊, ⪋, ⪌, ⪍, ⪎, ⪏, ⪐, ⪑, ⪒, ⪓, ⪔, ⪕, ⪖, ⪗, ⪘, ⪙, ⪚, ⪛, ⪜, ⪝, ⪞, ⪟, ⪠, ⪡, ⪢, ⪣, ⪤, ⪥, ⪦, ⪧, ⪨, ⪩, ⪪, ⪫, ⪬, ⪭, ⪮, ⪯, ⪰, ⪱, ⪲, ⪳, ⪴, ⪵, ⪶, ⪷, ⪸, ⪹, ⪺, ⪻, ⪼, ⪽, ⪾, ⪿, ⫀, ⫁, ⫂, ⫃, ⫄, ⫅, ⫆, ⫇, ⫈, ⫉, ⫊, ⫋, ⫌, ⫍, ⫎, ⫏, ⫐, ⫑, ⫒, ⫓, ⫔, ⫕, ⫖, ⫗, ⫘, ⫙, ⫚, ⫛, ⫝̸, ⫬, ⫭, ⫮, ⫯, ⫰, ⫱, ⫲, ⫳, ⫽, ⫾, ⫿, 🈑

form:postfix lspace:0 rspace:0

Priority: 100

’, ”

Priority: 120

),], |, ||, }, ‖, ⌉, ⌋, 〉, ❭, ⟩, ⟫, ⟭, ⟯, ❩, ❫, ❯, ❱, ❳, ❵, ⸣, ⸥, ⸧, ⸩, 〉

Priority: 820

!, !!, %, ′

Priority: 920

", &, ', ++, --, ^, _, `, ~, ¨, ¯, °, ², ³, ´, ¸, ¹, ˆ, ˇ, ˉ, ˊ, ˋ, ˍ, ˘, ˙, ˚, ˜, ˝, ˷, ̂, ̑, ‗, ‾, ‴, ⁗, ⃛, ⃜, ⏜, ⏝, ⏞, ⏟, ⏠, ⏡, ﹍, ﹎, ﹏

form:prefix lspace:0 rspace:0

Priority: 100

‘, “

Priority: 120

(, [, {, |, ||, ‖, ⌈, ⌊, 〈, ❬, ⟦, ⟨, ⟪, ⟬, ⟮, ❨, ❪, ❬, ❮, ❰, ❲, ❴, ⸢, ⸤, ⸦, ⸨, 〈

Priority: 200

∴, ∵

Priority: 280

!, ¬, ∀, ∃, ∄, ∼, ⌐, ⌙, ⫬, ⫭

Priority: 580

∟, ∠, ∡, ∢, ⊾, ⊿, ⟀, ⦛, ⦜, ⦝, ⦞, ⦟, ⦠, ⦡, ⦢, ⦣, ⦤, ⦥, ⦦, ⦧, ⦨, ⦩, ⦪, ⦫, ⦬, ⦭, ⦮, ⦯

Priority: 720

+, -, ±, ∁, −, ∓, ➕, ➖

Priority: 780

∇

form:prefix lspace:3 rspace:0

Priority: 780

ⅅ, ⅆ, ∂

Priority: 860

√, ∛, ∜

form:prefix lspace:3 rspace:3

Priority: 440

∑, ⨊, ⨋, ⨝, ⨞

Priority: 460

⨁

Priority: 480

∫, ∬, ∭, ∮, ∯, ∰, ∱, ∲, ∳, ⨌, ⨍, ⨎, ⨏, ⨐, ⨑, ⨒, ⨓, ⨔, ⨕, ⨖, ⨗, ⨘, ⨙, ⨚, ⨛, ⨜

Priority: 500

⋃, ⨃, ⨄

Priority: 520

⋀, ⋁, ⋂, ⨀, ⨂, ⨅, ⨆, ⨇, ⨈, ⨉, ⫼, ⫿

Priority: 540

∏, ∐

B.3.2 Sortable Table View

► Show Section

Character	Glyph	Name	form	priority	lspace	rspace	Properties
‘	‘	left single quotation mark	**prefix**	100	0	0	
’	’	right single quotation mark	**postfix**	100	0	0	
“	“	left double quotation mark	**prefix**	100	0	0	
”	”	right double quotation mark	**postfix**	100	0	0	
(	(	left parenthesis	**prefix**	120	0	0	stretchy, symmetric
)	)	right parenthesis	**postfix**	120	0	0	stretchy, symmetric
[	[	left square bracket	**prefix**	120	0	0	stretchy, symmetric
]	]	right square bracket	**postfix**	120	0	0	stretchy, symmetric
{	{	left curly bracket	**prefix**	120	0	0	stretchy, symmetric
|	|	vertical line	**prefix**	120	0	0	stretchy, symmetric
|	|	vertical line	**postfix**	120	0	0	stretchy, symmetric

Character	Glyph	Name	form	priority	lspace	rspace	Properties
‖	‖	multiple character operator: ||	**prefix**	120	0	0	
‖	‖	multiple character operator: ||	**postfix**	120	0	0	
}	}	right curly bracket	**postfix**	120	0	0	stretchy, symmetric
&#x2016;	‖	double vertical line	**prefix**	120	0	0	stretchy, symmetric
&#x2016;	‖	double vertical line	**postfix**	120	0	0	stretchy, symmetric
&#x2308;	⌈	left ceiling	**prefix**	120	0	0	stretchy, symmetric
&#x2309;	⌉	right ceiling	**postfix**	120	0	0	stretchy, symmetric
&#x230A;	⌊	left floor	**prefix**	120	0	0	stretchy, symmetric
&#x230B;	⌋	right floor	**postfix**	120	0	0	stretchy, symmetric
&#x2329;	〈	left-pointing angle bracket	**prefix**	120	0	0	stretchy, symmetric
&#x232A;	〉	right-pointing angle bracket	**postfix**	120	0	0	stretchy, symmetric
&#x2772;	❲	light left tortoise shell bracket ornament	**prefix**	120	0	0	stretchy, symmetric
&#x2773;	❳	light right tortoise shell bracket ornament	**postfix**	120	0	0	stretchy, symmetric
&#x27E6;	⟦	mathematical left white square bracket	**prefix**	120	0	0	stretchy, symmetric
&#x27E7;	⟧	mathematical right white square bracket	**postfix**	120	0	0	stretchy, symmetric
&#x27E8;	⟨	mathematical left angle bracket	**prefix**	120	0	0	stretchy, symmetric
&#x27E9;	⟩	mathematical right angle bracket	**postfix**	120	0	0	stretchy, symmetric
&#x27EA;	⟪	mathematical left double angle bracket	**prefix**	120	0	0	stretchy, symmetric
&#x27EB;	⟫	mathematical right double angle bracket	**postfix**	120	0	0	stretchy, symmetric
&#x27EC;	⟬	mathematical left white tortoise shell bracket	**prefix**	120	0	0	stretchy, symmetric
&#x27ED;	⟭	mathematical right white tortoise shell bracket	**postfix**	120	0	0	stretchy, symmetric
&#x27EE;	⟮	mathematical left flattened parenthesis	**prefix**	120	0	0	stretchy, symmetric
&#x27EF;	⟯	mathematical right flattened parenthesis	**postfix**	120	0	0	stretchy, symmetric
&#x2980;	⦀	triple vertical bar delimiter	**prefix**	120	0	0	stretchy, symmetric
&#x2980;	⦀	triple vertical bar delimiter	**postfix**	120	0	0	stretchy, symmetric
&#x2983;	⦃	left white curly bracket	**prefix**	120	0	0	stretchy, symmetric
&#x2984;	⦄	right white curly bracket	**postfix**	120	0	0	stretchy, symmetric
&#x2985;	⦅	left white parenthesis	**prefix**	120	0	0	stretchy, symmetric
&#x2986;	⦆	right white parenthesis	**postfix**	120	0	0	stretchy, symmetric
&#x2987;	⦇	z notation left image bracket	**prefix**	120	0	0	stretchy, symmetric
&#x2988;	⦈	z notation right image bracket	**postfix**	120	0	0	stretchy, symmetric
&#x2989;	⦉	z notation left binding bracket	**prefix**	120	0	0	stretchy, symmetric
&#x298A;	⦊	z notation right binding bracket	**postfix**	120	0	0	stretchy, symmetric
&#x298B;	⦋	left square bracket with underbar	**prefix**	120	0	0	stretchy, symmetric
&#x298C;	⦌	right square bracket with underbar	**postfix**	120	0	0	stretchy, symmetric
&#x298D;	⦍	left square bracket with tick in top corner	**prefix**	120	0	0	stretchy, symmetric
&#x298E;	⦎	right square bracket with tick in bottom corner	**postfix**	120	0	0	stretchy, symmetric
&#x298F;	⦏	left square bracket with tick in bottom corner	**prefix**	120	0	0	stretchy, symmetric
&#x2990;	⦐	right square bracket with tick in top corner	**postfix**	120	0	0	stretchy, symmetric
&#x2991;	⦑	left angle bracket with dot	**prefix**	120	0	0	stretchy, symmetric

Character	Glyph	Name	form	priority	lspace	rspace	Properties
⦒	⦒	right angle bracket with dot	postfix	120	0	0	stretchy, symmetric
⦓	⦓	left arc less-than bracket	prefix	120	0	0	stretchy, symmetric
⦔	⦔	right arc greater-than bracket	postfix	120	0	0	stretchy, symmetric
⦕	⦕	double left arc greater-than bracket	prefix	120	0	0	stretchy, symmetric
⦖	⦖	double right arc less-than bracket	postfix	120	0	0	stretchy, symmetric
⦗	⦗	left black tortoise shell bracket	prefix	120	0	0	stretchy, symmetric
⦘	⦘	right black tortoise shell bracket	postfix	120	0	0	stretchy, symmetric
⦙	⦙	dotted fence	prefix	120	0	0	stretchy, symmetric
⦙	⦙	dotted fence	postfix	120	0	0	stretchy, symmetric
⧘	⧘	left wiggly fence	prefix	120	0	0	stretchy, symmetric
⧙	⧙	right wiggly fence	postfix	120	0	0	stretchy, symmetric
⧚	⧚	left double wiggly fence	prefix	120	0	0	stretchy, symmetric
⧛	⧛	right double wiggly fence	postfix	120	0	0	stretchy, symmetric
⧼	⧼	left-pointing curved angle bracket	prefix	120	0	0	stretchy, symmetric
⧽	⧽	right-pointing curved angle bracket	postfix	120	0	0	stretchy, symmetric
;	;	semicolon	infix	140	0	3	linebreakstyle=after
⦁	⦁	z notation spot	infix	140	5	5	
,	,	comma	infix	160	0	3	linebreakstyle=after
⁣	⁣	invisible separator	infix	160	0	0	linebreakstyle=after
:	:	colon	infix	180	0	3	
⦂	⦂	z notation type colon	infix	180	5	5	
∴	∴	therefore	prefix	200	0	0	
∵	∵	because	prefix	200	0	0	
->	->	multiple character operator: ->	infix	220	5	5	
⊶	⊶	original of	infix	220	5	5	
⊷	⊷	image of	infix	220	5	5	
⊸	⊸	multimap	infix	220	5	5	
⧴	⧴	rule-delayed	infix	220	5	5	
//	//	multiple character operator: //	infix	240	5	5	
⊢	⊢	right tack	infix	260	5	5	
⊣	⊣	left tack	infix	260	5	5	
⊧	⊧	models	infix	260	5	5	
⊨	⊨	true	infix	260	5	5	
⊩	⊩	forces	infix	260	5	5	
⊪	⊪	triple vertical bar right turnstile	infix	260	5	5	
⊫	⊫	double vertical bar double right turnstile	infix	260	5	5	
⊬	⊬	does not prove	infix	260	5	5	
⊭	⊭	not true	infix	260	5	5	
⊮	⊮	does not force	infix	260	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x22AF;	⊯	negated double vertical bar double right turnstile	infix	260	5	5	
&#x2ADE;	⫞	short left tack	infix	260	5	5	
&#x2ADF;	⫟	short down tack	infix	260	5	5	
&#x2AE0;	⫠	short up tack	infix	260	5	5	
&#x2AE1;	⫡	perpendicular with s	infix	260	5	5	
&#x2AE2;	⫢	vertical bar triple right turnstile	infix	260	5	5	
&#x2AE3;	⫣	double vertical bar left turnstile	infix	260	5	5	
&#x2AE4;	⫤	vertical bar double left turnstile	infix	260	5	5	
&#x2AE5;	⫥	double vertical bar double left turnstile	infix	260	5	5	
&#x2AE6;	⫦	long dash from left member of double vertical	infix	260	5	5	
&#x2AE7;	⫧	short down tack with overbar	infix	260	5	5	
&#x2AE8;	⫨	short up tack with underbar	infix	260	5	5	
&#x2AE9;	⫩	short up tack above short down tack	infix	260	5	5	
&#x2AEA;	⫪	double down tack	infix	260	5	5	
&#x2AEB;	⫫	double up tack	infix	260	5	5	
!	!	exclamation mark	prefix	280	0	0	
&#xAC;	¬	not sign	prefix	280	0	0	
&#x2200;	∀	for all	prefix	280	0	0	
&#x2203;	∃	there exists	prefix	280	0	0	
&#x2204;	∄	there does not exist	prefix	280	0	0	
&#x223C;	∼	tilde operator	prefix	280	0	0	
&#x2310;	⌐	reversed not sign	prefix	280	0	0	
&#x2319;	⌙	turned not sign	prefix	280	0	0	
&#x2AEC;	⫬	double stroke not sign	prefix	280	0	0	
&#x2AED;	⫭	reversed double stroke not sign	prefix	280	0	0	
&#x2208;	∈	element of	infix	300	5	5	
&#x2209;	∉	not an element of	infix	300	5	5	
&#x220A;	∊	small element of	infix	300	5	5	
&#x220B;	∋	contains as member	infix	300	5	5	
&#x220C;	∌	does not contain as member	infix	300	5	5	
&#x220D;	∍	small contains as member	infix	300	5	5	
&#x2282;	⊂	subset of	infix	300	5	5	
&#x2283;	⊃	superset of	infix	300	5	5	
&#x2284;	⊄	not a subset of	infix	300	5	5	
&#x2285;	⊅	not a superset of	infix	300	5	5	
&#x2286;	⊆	subset of or equal to	infix	300	5	5	
&#x2287;	⊇	superset of or equal to	infix	300	5	5	
&#x2288;	⊈	neither a subset of nor equal to	infix	300	5	5	
&#x2289;	⊉	neither a superset of nor equal to	infix	300	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x228A;	⊊	subset of with not equal to	infix	300	5	5	
&#x228B;	⊋	superset of with not equal to	infix	300	5	5	
&#x228F;	⊏	square image of	infix	300	5	5	
&#x2290;	⊐	square original of	infix	300	5	5	
&#x2291;	⊑	square image of or equal to	infix	300	5	5	
&#x2292;	⊒	square original of or equal to	infix	300	5	5	
&#x22B0;	⊰	precedes under relation	infix	300	5	5	
&#x22B1;	⊱	succeeds under relation	infix	300	5	5	
&#x22B2;	⊲	normal subgroup of	infix	300	5	5	
&#x22B3;	⊳	contains as normal subgroup	infix	300	5	5	
&#x22D0;	⋐	double subset	infix	300	5	5	
&#x22D1;	⋑	double superset	infix	300	5	5	
&#x22E2;	⋢	not square image of or equal to	infix	300	5	5	
&#x22E3;	⋣	not square original of or equal to	infix	300	5	5	
&#x22E4;	⋤	square image of or not equal to	infix	300	5	5	
&#x22E5;	⋥	square original of or not equal to	infix	300	5	5	
&#x22EA;	⋪	not normal subgroup of	infix	300	5	5	
&#x22EB;	⋫	does not contain as normal subgroup	infix	300	5	5	
&#x22EC;	⋬	not normal subgroup of or equal to	infix	300	5	5	
&#x22ED;	⋭	does not contain as normal subgroup or equal	infix	300	5	5	
&#x22F2;	⋲	element of with long horizontal stroke	infix	300	5	5	
&#x22F3;	⋳	element of with vertical bar at end of horizontal stroke	infix	300	5	5	
&#x22F4;	⋴	small element of with vertical bar at end of horizontal stroke	infix	300	5	5	
&#x22F5;	⋵	element of with dot above	infix	300	5	5	
&#x22F6;	⋶	element of with overbar	infix	300	5	5	
&#x22F7;	⋷	small element of with overbar	infix	300	5	5	
&#x22F8;	⋸	element of with underbar	infix	300	5	5	
&#x22F9;	⋹	element of with two horizontal strokes	infix	300	5	5	
&#x22FA;	⋺	contains with long horizontal stroke	infix	300	5	5	
&#x22FB;	⋻	contains with vertical bar at end of horizontal stroke	infix	300	5	5	
&#x22FC;	⋼	small contains with vertical bar at end of horizontal stroke	infix	300	5	5	
&#x22FD;	⋽	contains with overbar	infix	300	5	5	
&#x22FE;	⋾	small contains with overbar	infix	300	5	5	
&#x22FF;	⋿	z notation bag membership	infix	300	5	5	
&#x2979;	⥹	subset above rightwards arrow	infix	300	5	5	
&#x297A;	⥺	leftwards arrow through subset	infix	300	5	5	
&#x297B;	⥻	superset above leftwards arrow	infix	300	5	5	
&#x2ABD;	⪽	subset with dot	infix	300	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
⪾	⪾	superset with dot	infix	300	5	5	
⪿	⪿	subset with plus sign below	infix	300	5	5	
⫀	⫀	superset with plus sign below	infix	300	5	5	
⫁	⫁	subset with multiplication sign below	infix	300	5	5	
⫂	⫂	superset with multiplication sign below	infix	300	5	5	
⫃	⫃	subset of or equal to with dot above	infix	300	5	5	
⫄	⫄	superset of or equal to with dot above	infix	300	5	5	
⫅	⫅	subset of above equals sign	infix	300	5	5	
⫆	⫆	superset of above equals sign	infix	300	5	5	
⫇	⫇	subset of above tilde operator	infix	300	5	5	
⫈	⫈	superset of above tilde operator	infix	300	5	5	
⫉	⫉	subset of above almost equal to	infix	300	5	5	
⫊	⫊	superset of above almost equal to	infix	300	5	5	
⫋	⫋	subset of above not equal to	infix	300	5	5	
⫌	⫌	superset of above not equal to	infix	300	5	5	
⫍	⫍	square left open box operator	infix	300	5	5	
⫎	⫎	square right open box operator	infix	300	5	5	
⫏	⫏	closed subset	infix	300	5	5	
⫐	⫐	closed superset	infix	300	5	5	
⫑	⫑	closed subset or equal to	infix	300	5	5	
⫒	⫒	closed superset or equal to	infix	300	5	5	
⫓	⫓	subset above superset	infix	300	5	5	
⫔	⫔	superset above subset	infix	300	5	5	
⫕	⫕	subset above subset	infix	300	5	5	
⫖	⫖	superset above superset	infix	300	5	5	
⫗	⫗	superset beside subset	infix	300	5	5	
⫘	⫘	superset beside and joined by dash with subset	infix	300	5	5	
⫙	⫙	element of opening downwards	infix	300	5	5	
!=	!=	multiple character operator: !=	infix	320	5	5	
*=	*=	multiple character operator: *=	infix	320	5	5	
+=	+=	multiple character operator: +=	infix	320	5	5	
-=	-=	multiple character operator: -=	infix	320	5	5	
/=	/=	multiple character operator: /=	infix	320	5	5	
:=	:=	multiple character operator: :=	infix	320	5	5	
<	<	less-than sign	infix	320	5	5	
<=	<=	multiple character operator: <=	infix	320	5	5	
=	=	equals sign	infix	320	5	5	
==	==	multiple character operator: ==	infix	320	5	5	
>	>	greater-than sign	infix	320	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
>=	≥	multiple character operator: >=	infix	320	5	5	
∣	|	vertical line	infix	320	5	5	
∥	||	multiple character operator: ||	infix	320	5	5	
∝	∝	proportional to	infix	320	5	5	
∣	∣	divides	infix	320	5	5	
∤	∤	does not divide	infix	320	5	5	
∥	∥	parallel to	infix	320	5	5	
∦	∦	not parallel to	infix	320	5	5	
∷	∷	proportion	infix	320	5	5	
∹	∹	excess	infix	320	5	5	
∺	∺	geometric proportion	infix	320	5	5	
∻	∻	homothetic	infix	320	5	5	
∼	∼	tilde operator	infix	320	5	5	
∽	∽	reversed tilde	infix	320	5	5	
∾	∾	inverted lazy s	infix	320	5	5	
≁	≁	not tilde	infix	320	5	5	
≂	≂	minus tilde	infix	320	5	5	
≃	≃	asymptotically equal to	infix	320	5	5	
≄	≄	not asymptotically equal to	infix	320	5	5	
≅	≅	approximately equal to	infix	320	5	5	
≆	≆	approximately but not actually equal to	infix	320	5	5	
≇	≇	neither approximately nor actually equal to	infix	320	5	5	
≈	≈	almost equal to	infix	320	5	5	
≉	≉	not almost equal to	infix	320	5	5	
≊	≊	almost equal or equal to	infix	320	5	5	
≋	≋	triple tilde	infix	320	5	5	
≌	≌	all equal to	infix	320	5	5	
≍	≍	equivalent to	infix	320	5	5	
≎	≎	geometrically equivalent to	infix	320	5	5	
≏	≏	difference between	infix	320	5	5	
≐	≐	approaches the limit	infix	320	5	5	
≑	≑	geometrically equal to	infix	320	5	5	
≒	≒	approximately equal to or the image of	infix	320	5	5	
≓	≓	image of or approximately equal to	infix	320	5	5	
≔	≔	colon equals	infix	320	5	5	
≕	≕	equals colon	infix	320	5	5	
≖	≖	ring in equal to	infix	320	5	5	
≗	≗	ring equal to	infix	320	5	5	
≘	≘	corresponds to	infix	320	5	5	
≙	≙	estimates	infix	320	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x225A;	≚	equiangular to	infix	320	5	5	
&#x225B;	≛	star equals	infix	320	5	5	
&#x225C;	≜	delta equal to	infix	320	5	5	
&#x225D;	≝	equal to by definition	infix	320	5	5	
&#x225E;	≞	measured by	infix	320	5	5	
&#x225F;	≟	questioned equal to	infix	320	5	5	
&#x2260;	≠	not equal to	infix	320	5	5	
&#x2261;	≡	identical to	infix	320	5	5	
&#x2262;	≢	not identical to	infix	320	5	5	
&#x2263;	≣	strictly equivalent to	infix	320	5	5	
&#x2264;	≤	less-than or equal to	infix	320	5	5	
&#x2265;	≥	greater-than or equal to	infix	320	5	5	
&#x2266;	≦	less-than over equal to	infix	320	5	5	
&#x2267;	≧	greater-than over equal to	infix	320	5	5	
&#x2268;	≨	less-than but not equal to	infix	320	5	5	
&#x2269;	≩	greater-than but not equal to	infix	320	5	5	
&#x226A;	≪	much less-than	infix	320	5	5	
&#x226B;	≫	much greater-than	infix	320	5	5	
&#x226C;	≬	between	infix	320	5	5	
&#x226D;	≭	not equivalent to	infix	320	5	5	
&#x226E;	≮	not less-than	infix	320	5	5	
&#x226F;	≯	not greater-than	infix	320	5	5	
&#x2270;	≰	neither less-than nor equal to	infix	320	5	5	
&#x2271;	≱	neither greater-than nor equal to	infix	320	5	5	
&#x2272;	≲	less-than or equivalent to	infix	320	5	5	
&#x2273;	≳	greater-than or equivalent to	infix	320	5	5	
&#x2274;	≴	neither less-than nor equivalent to	infix	320	5	5	
&#x2275;	≵	neither greater-than nor equivalent to	infix	320	5	5	
&#x2276;	≶	less-than or greater-than	infix	320	5	5	
&#x2277;	≷	greater-than or less-than	infix	320	5	5	
&#x2278;	≸	neither less-than nor greater-than	infix	320	5	5	
&#x2279;	≹	neither greater-than nor less-than	infix	320	5	5	
&#x227A;	≺	precedes	infix	320	5	5	
&#x227B;	≻	succeeds	infix	320	5	5	
&#x227C;	≼	precedes or equal to	infix	320	5	5	
&#x227D;	≽	succeeds or equal to	infix	320	5	5	
&#x227E;	≾	precedes or equivalent to	infix	320	5	5	
&#x227F;	≿	succeeds or equivalent to	infix	320	5	5	
&#x2280;	⊀	does not precede	infix	320	5	5	
&#x2281;	⊁	does not succeed	infix	320	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x229C;	⊜	circled equals	infix	320	5	5	
&#x22A6;	⊦	assertion	infix	320	5	5	
&#x22B4;	⊴	normal subgroup of or equal to	infix	320	5	5	
&#x22B5;	⊵	contains as normal subgroup or equal to	infix	320	5	5	
&#x22C8;	⋈	bowtie	infix	320	5	5	
&#x22CD;	⋍	reversed tilde equals	infix	320	5	5	
&#x22D4;	⋔	pitchfork	infix	320	5	5	
&#x22D5;	⋕	equal and parallel to	infix	320	5	5	
&#x22D6;	⋖	less-than with dot	infix	320	5	5	
&#x22D7;	⋗	greater-than with dot	infix	320	5	5	
&#x22D8;	⋘	very much less-than	infix	320	5	5	
&#x22D9;	⋙	very much greater-than	infix	320	5	5	
&#x22DA;	⋚	less-than equal to or greater-than	infix	320	5	5	
&#x22DB;	⋛	greater-than equal to or less-than	infix	320	5	5	
&#x22DC;	⋜	equal to or less-than	infix	320	5	5	
&#x22DD;	⋝	equal to or greater-than	infix	320	5	5	
&#x22DE;	⋞	equal to or precedes	infix	320	5	5	
&#x22DF;	⋟	equal to or succeeds	infix	320	5	5	
&#x22E0;	⋠	does not precede or equal	infix	320	5	5	
&#x22E1;	⋡	does not succeed or equal	infix	320	5	5	
&#x22E6;	⋦	less-than but not equivalent to	infix	320	5	5	
&#x22E7;	⋧	greater-than but not equivalent to	infix	320	5	5	
&#x22E8;	⋨	precedes but not equivalent to	infix	320	5	5	
&#x22E9;	⋩	succeeds but not equivalent to	infix	320	5	5	
&#x27C2;	⟂	perpendicular	infix	320	5	5	
&#x2976;	⥶	less-than above leftwards arrow	infix	320	5	5	
&#x2977;	⥷	leftwards arrow through less-than	infix	320	5	5	
&#x2978;	⥸	greater-than above rightwards arrow	infix	320	5	5	
&#x29B6;	⦶	circled vertical bar	infix	320	5	5	
&#x29B7;	⦷	circled parallel	infix	320	5	5	
&#x29B9;	⦹	circled perpendicular	infix	320	5	5	
&#x29C0;	⧀	circled less-than	infix	320	5	5	
&#x29C1;	⧁	circled greater-than	infix	320	5	5	
&#x29CE;	⧎	right triangle above left triangle	infix	320	5	5	
&#x29CF;	⧏	left triangle beside vertical bar	infix	320	5	5	
&#x29D0;	⧐	vertical bar beside right triangle	infix	320	5	5	
&#x29D1;	⧑	bowtie with left half black	infix	320	5	5	
&#x29D2;	⧒	bowtie with right half black	infix	320	5	5	
&#x29D3;	⧓	black bowtie	infix	320	5	5	
&#x29E1;	⧡	increases as	infix	320	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x29E3;	⧣	equals sign and slanted parallel	infix	320	5	5	
&#x29E4;	⧤	equals sign and slanted parallel with tilde above	infix	320	5	5	
&#x29E5;	⧥	identical to and slanted parallel	infix	320	5	5	
&#x29E6;	⧦	gleich stark	infix	320	5	5	
&#x2A66;	⩦	equals sign with dot below	infix	320	5	5	
&#x2A67;	⩧	identical with dot above	infix	320	5	5	
&#x2A68;	⩨	triple horizontal bar with double vertical stroke	infix	320	5	5	
&#x2A69;	⩩	triple horizontal bar with triple vertical stroke	infix	320	5	5	
&#x2A6A;	⩪	tilde operator with dot above	infix	320	5	5	
&#x2A6B;	⩫	tilde operator with rising dots	infix	320	5	5	
&#x2A6C;	⩬	similar minus similar	infix	320	5	5	
&#x2A6D;	⩭	congruent with dot above	infix	320	5	5	
&#x2A6E;	⩮	equals with asterisk	infix	320	5	5	
&#x2A6F;	⩯	almost equal to with circumflex accent	infix	320	5	5	
&#x2A70;	⩰	approximately equal or equal to	infix	320	5	5	
&#x2A71;	⩱	equals sign above plus sign	infix	320	5	5	
&#x2A72;	⩲	plus sign above equals sign	infix	320	5	5	
&#x2A73;	⩳	equals sign above tilde operator	infix	320	5	5	
&#x2A74;	⩴	double colon equal	infix	320	5	5	
&#x2A75;	⩵	two consecutive equals signs	infix	320	5	5	
&#x2A76;	⩶	three consecutive equals signs	infix	320	5	5	
&#x2A77;	⩷	equals sign with two dots above and two dots below	infix	320	5	5	
&#x2A78;	⩸	equivalent with four dots above	infix	320	5	5	
&#x2A79;	⩹	less-than with circle inside	infix	320	5	5	
&#x2A7A;	⩺	greater-than with circle inside	infix	320	5	5	
&#x2A7B;	⩻	less-than with question mark above	infix	320	5	5	
&#x2A7C;	⩼	greater-than with question mark above	infix	320	5	5	
&#x2A7D;	⩽	less-than or slanted equal to	infix	320	5	5	
&#x2A7E;	⩾	greater-than or slanted equal to	infix	320	5	5	
&#x2A7F;	⩿	less-than or slanted equal to with dot inside	infix	320	5	5	
&#x2A80;	⪀	greater-than or slanted equal to with dot inside	infix	320	5	5	
&#x2A81;	⪁	less-than or slanted equal to with dot above	infix	320	5	5	
&#x2A82;	⪂	greater-than or slanted equal to with dot above	infix	320	5	5	
&#x2A83;	⪃	less-than or slanted equal to with dot above right	infix	320	5	5	
&#x2A84;	⪄	greater-than or slanted equal to with dot above left	infix	320	5	5	
&#x2A85;	⪅	less-than or approximate	infix	320	5	5	
&#x2A86;	⪆	greater-than or approximate	infix	320	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2A87;	⪇	less-than and single-line not equal to	infix	320	5	5	
&#x2A88;	⪈	greater-than and single-line not equal to	infix	320	5	5	
&#x2A89;	⪉	less-than and not approximate	infix	320	5	5	
&#x2A8A;	⪊	greater-than and not approximate	infix	320	5	5	
&#x2A8B;	⪋	less-than above double-line equal above greater-than	infix	320	5	5	
&#x2A8C;	⪌	greater-than above double-line equal above less-than	infix	320	5	5	
&#x2A8D;	⪍	less-than above similar or equal	infix	320	5	5	
&#x2A8E;	⪎	greater-than above similar or equal	infix	320	5	5	
&#x2A8F;	⪏	less-than above similar above greater-than	infix	320	5	5	
&#x2A90;	⪐	greater-than above similar above less-than	infix	320	5	5	
&#x2A91;	⪑	less-than above greater-than above double-line equal	infix	320	5	5	
&#x2A92;	⪒	greater-than above less-than above double-line equal	infix	320	5	5	
&#x2A93;	⪓	less-than above slanted equal above greater-than above slanted equal	infix	320	5	5	
&#x2A94;	⪔	greater-than above slanted equal above less-than above slanted equal	infix	320	5	5	
&#x2A95;	⪕	slanted equal to or less-than	infix	320	5	5	
&#x2A96;	⪖	slanted equal to or greater-than	infix	320	5	5	
&#x2A97;	⪗	slanted equal to or less-than with dot inside	infix	320	5	5	
&#x2A98;	⪘	slanted equal to or greater-than with dot inside	infix	320	5	5	
&#x2A99;	⪙	double-line equal to or less-than	infix	320	5	5	
&#x2A9A;	⪚	double-line equal to or greater-than	infix	320	5	5	
&#x2A9B;	⪛	double-line slanted equal to or less-than	infix	320	5	5	
&#x2A9C;	⪜	double-line slanted equal to or greater-than	infix	320	5	5	
&#x2A9D;	⪝	similar or less-than	infix	320	5	5	
&#x2A9E;	⪞	similar or greater-than	infix	320	5	5	
&#x2A9F;	⪟	similar above less-than above equals sign	infix	320	5	5	
&#x2AA0;	⪠	similar above greater-than above equals sign	infix	320	5	5	
&#x2AA1;	⪡	double nested less-than	infix	320	5	5	
&#x2AA2;	⪢	double nested greater-than	infix	320	5	5	
&#x2AA3;	⪣	double nested less-than with underbar	infix	320	5	5	
&#x2AA4;	⪤	greater-than overlapping less-than	infix	320	5	5	
&#x2AA5;	⪥	greater-than beside less-than	infix	320	5	5	
&#x2AA6;	⪦	less-than closed by curve	infix	320	5	5	
&#x2AA7;	⪧	greater-than closed by curve	infix	320	5	5	
&#x2AA8;	⪨	less-than closed by curve above slanted equal	infix	320	5	5	
&#x2AA9;	⪩	greater-than closed by curve above slanted equal	infix	320	5	5	
&#x2AAA;	⪪	smaller than	infix	320	5	5	
&#x2AAB;	⪫	larger than	infix	320	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2AAC;	⪬	smaller than or equal to	infix	320	5	5	
&#x2AAD;	⪭	larger than or equal to	infix	320	5	5	
&#x2AAE;	⪮	equals sign with bumpy above	infix	320	5	5	
&#x2AAF;	⪯	precedes above single-line equals sign	infix	320	5	5	
&#x2AB0;	⪰	succeeds above single-line equals sign	infix	320	5	5	
&#x2AB1;	⪱	precedes above single-line not equal to	infix	320	5	5	
&#x2AB2;	⪲	succeeds above single-line not equal to	infix	320	5	5	
&#x2AB3;	⪳	precedes above equals sign	infix	320	5	5	
&#x2AB4;	⪴	succeeds above equals sign	infix	320	5	5	
&#x2AB5;	⪵	precedes above not equal to	infix	320	5	5	
&#x2AB6;	⪶	succeeds above not equal to	infix	320	5	5	
&#x2AB7;	⪷	precedes above almost equal to	infix	320	5	5	
&#x2AB8;	⪸	succeeds above almost equal to	infix	320	5	5	
&#x2AB9;	⪹	precedes above not almost equal to	infix	320	5	5	
&#x2ABA;	⪺	succeeds above not almost equal to	infix	320	5	5	
&#x2ABB;	⪻	double precedes	infix	320	5	5	
&#x2ABC;	⪼	double succeeds	infix	320	5	5	
&#x2ADA;	⫚	pitchfork with tee top	infix	320	5	5	
&#x2AEE;	⫮	does not divide with reversed negation slash	infix	320	5	5	
&#x2AF2;	⫲	parallel with horizontal stroke	infix	320	5	5	
&#x2AF3;	⫳	parallel with tilde operator	infix	320	5	5	
&#x2AF4;	⫴	triple vertical bar binary relation	infix	320	5	5	
&#x2AF5;	⫵	triple vertical bar with horizontal stroke	infix	320	5	5	
&#x2AF7;	⫷	triple nested less-than	infix	320	5	5	
&#x2AF8;	⫸	triple nested greater-than	infix	320	5	5	
&#x2AF9;	⫹	double-line slanted less-than or equal to	infix	320	5	5	
&#x2AFA;	⫺	double-line slanted greater-than or equal to	infix	320	5	5	
&#x2BD1;	⯑	uncertainty sign	infix	320	5	5	
&#x2190;	←	leftwards arrow	infix	340	5	5	stretchy
&#x2191;	↑	upwards arrow	infix	340	5	5	stretchy
&#x2192;	→	rightwards arrow	infix	340	5	5	stretchy
&#x2193;	↓	downwards arrow	infix	340	5	5	stretchy
&#x2194;	↔	left right arrow	infix	340	5	5	stretchy
&#x2195;	↕	up down arrow	infix	340	5	5	stretchy
&#x2196;	↖	north west arrow	infix	340	5	5	
&#x2197;	↗	north east arrow	infix	340	5	5	
&#x2198;	↘	south east arrow	infix	340	5	5	
&#x2199;	↙	south west arrow	infix	340	5	5	
&#x219A;	↚	leftwards arrow with stroke	infix	340	5	5	stretchy
&#x219B;	↛	rightwards arrow with stroke	infix	340	5	5	stretchy

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x219C;	↜	leftwards wave arrow	infix	340	5	5	stretchy
&#x219D;	↝	rightwards wave arrow	infix	340	5	5	stretchy
&#x219E;	↞	leftwards two headed arrow	infix	340	5	5	stretchy
&#x219F;	↟	upwards two headed arrow	infix	340	5	5	stretchy
&#x21A0;	↠	rightwards two headed arrow	infix	340	5	5	stretchy
&#x21A1;	↡	downwards two headed arrow	infix	340	5	5	stretchy
&#x21A2;	↢	leftwards arrow with tail	infix	340	5	5	stretchy
&#x21A3;	↣	rightwards arrow with tail	infix	340	5	5	stretchy
&#x21A4;	↤	leftwards arrow from bar	infix	340	5	5	stretchy
&#x21A5;	↥	upwards arrow from bar	infix	340	5	5	stretchy
&#x21A6;	↦	rightwards arrow from bar	infix	340	5	5	stretchy
&#x21A7;	↧	downwards arrow from bar	infix	340	5	5	stretchy
&#x21A8;	↨	up down arrow with base	infix	340	5	5	stretchy
&#x21A9;	↩	leftwards arrow with hook	infix	340	5	5	stretchy
&#x21AA;	↪	rightwards arrow with hook	infix	340	5	5	stretchy
&#x21AB;	↫	leftwards arrow with loop	infix	340	5	5	stretchy
&#x21AC;	↬	rightwards arrow with loop	infix	340	5	5	stretchy
&#x21AD;	↭	left right wave arrow	infix	340	5	5	stretchy
&#x21AE;	↮	left right arrow with stroke	infix	340	5	5	stretchy
&#x21AF;	↯	downwards zigzag arrow	infix	340	5	5	
&#x21B0;	↰	upwards arrow with tip leftwards	infix	340	5	5	stretchy
&#x21B1;	↱	upwards arrow with tip rightwards	infix	340	5	5	stretchy
&#x21B2;	↲	downwards arrow with tip leftwards	infix	340	5	5	stretchy
&#x21B3;	↳	downwards arrow with tip rightwards	infix	340	5	5	stretchy
&#x21B4;	↴	rightwards arrow with corner downwards	infix	340	5	5	stretchy
&#x21B5;	↵	downwards arrow with corner leftwards	infix	340	5	5	stretchy
&#x21B6;	↶	anticlockwise top semicircle arrow	infix	340	5	5	
&#x21B7;	↷	clockwise top semicircle arrow	infix	340	5	5	
&#x21B8;	↸	north west arrow to long bar	infix	340	5	5	
&#x21B9;	↹	leftwards arrow to bar over rightwards arrow to bar	infix	340	5	5	stretchy
&#x21BA;	↺	anticlockwise open circle arrow	infix	340	5	5	
&#x21BB;	↻	clockwise open circle arrow	infix	340	5	5	
&#x21BC;	↼	leftwards harpoon with barb upwards	infix	340	5	5	stretchy
&#x21BD;	↽	leftwards harpoon with barb downwards	infix	340	5	5	stretchy
&#x21BE;	↾	upwards harpoon with barb rightwards	infix	340	5	5	stretchy
&#x21BF;	↿	upwards harpoon with barb leftwards	infix	340	5	5	stretchy
&#x21C0;	⇀	rightwards harpoon with barb upwards	infix	340	5	5	stretchy
&#x21C1;	⇁	rightwards harpoon with barb downwards	infix	340	5	5	stretchy
&#x21C2;	⇂	downwards harpoon with barb rightwards	infix	340	5	5	stretchy

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x21C3;	⇃	downwards harpoon with barb leftwards	infix	340	5	5	stretchy
&#x21C4;	⇄	rightwards arrow over leftwards arrow	infix	340	5	5	stretchy
&#x21C5;	⇅	upwards arrow leftwards of downwards arrow	infix	340	5	5	stretchy
&#x21C6;	⇆	leftwards arrow over rightwards arrow	infix	340	5	5	stretchy
&#x21C7;	⇇	leftwards paired arrows	infix	340	5	5	stretchy
&#x21C8;	⇈	upwards paired arrows	infix	340	5	5	stretchy
&#x21C9;	⇉	rightwards paired arrows	infix	340	5	5	stretchy
&#x21CA;	⇊	downwards paired arrows	infix	340	5	5	stretchy
&#x21CB;	⇋	leftwards harpoon over rightwards harpoon	infix	340	5	5	stretchy
&#x21CC;	⇌	rightwards harpoon over leftwards harpoon	infix	340	5	5	stretchy
&#x21CD;	⇍	leftwards double arrow with stroke	infix	340	5	5	stretchy
&#x21CE;	⇎	left right double arrow with stroke	infix	340	5	5	stretchy
&#x21CF;	⇏	rightwards double arrow with stroke	infix	340	5	5	stretchy
&#x21D0;	⇐	leftwards double arrow	infix	340	5	5	stretchy
&#x21D1;	⇑	upwards double arrow	infix	340	5	5	stretchy
&#x21D2;	⇒	rightwards double arrow	infix	340	5	5	stretchy
&#x21D3;	⇓	downwards double arrow	infix	340	5	5	stretchy
&#x21D4;	⇔	left right double arrow	infix	340	5	5	stretchy
&#x21D5;	⇕	up down double arrow	infix	340	5	5	stretchy
&#x21D6;	⇖	north west double arrow	infix	340	5	5	
&#x21D7;	⇗	north east double arrow	infix	340	5	5	
&#x21D8;	⇘	south east double arrow	infix	340	5	5	
&#x21D9;	⇙	south west double arrow	infix	340	5	5	
&#x21DA;	⇚	leftwards triple arrow	infix	340	5	5	stretchy
&#x21DB;	⇛	rightwards triple arrow	infix	340	5	5	stretchy
&#x21DC;	⇜	leftwards squiggle arrow	infix	340	5	5	stretchy
&#x21DD;	⇝	rightwards squiggle arrow	infix	340	5	5	stretchy
&#x21DE;	⇞	upwards arrow with double stroke	infix	340	5	5	stretchy
&#x21DF;	⇟	downwards arrow with double stroke	infix	340	5	5	stretchy
&#x21E0;	⇠	leftwards dashed arrow	infix	340	5	5	stretchy
&#x21E1;	⇡	upwards dashed arrow	infix	340	5	5	stretchy
&#x21E2;	⇢	rightwards dashed arrow	infix	340	5	5	stretchy
&#x21E3;	⇣	downwards dashed arrow	infix	340	5	5	stretchy
&#x21E4;	⇤	leftwards arrow to bar	infix	340	5	5	stretchy
&#x21E5;	⇥	rightwards arrow to bar	infix	340	5	5	stretchy
&#x21E6;	⇦	leftwards white arrow	infix	340	5	5	stretchy
&#x21E7;	⇧	upwards white arrow	infix	340	5	5	stretchy
&#x21E8;	⇨	rightwards white arrow	infix	340	5	5	stretchy
&#x21E9;	⇩	downwards white arrow	infix	340	5	5	stretchy
&#x21EA;	⇪	upwards white arrow from bar	infix	340	5	5	stretchy

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x21EB;	⇫	upwards white arrow on pedestal	infix	340	5	5	stretchy
&#x21EC;	⇬	upwards white arrow on pedestal with horizontal bar	infix	340	5	5	stretchy
&#x21ED;	⇭	upwards white arrow on pedestal with vertical bar	infix	340	5	5	stretchy
&#x21EE;	⇮	upwards white double arrow	infix	340	5	5	stretchy
&#x21EF;	⇯	upwards white double arrow on pedestal	infix	340	5	5	stretchy
&#x21F0;	⇰	rightwards white arrow from wall	infix	340	5	5	stretchy
&#x21F1;	⇱	north west arrow to corner	infix	340	5	5	
&#x21F2;	⇲	south east arrow to corner	infix	340	5	5	
&#x21F3;	⇳	up down white arrow	infix	340	5	5	stretchy
&#x21F4;	⇴	right arrow with small circle	infix	340	5	5	stretchy
&#x21F5;	⇵	downwards arrow leftwards of upwards arrow	infix	340	5	5	stretchy
&#x21F6;	⇶	three rightwards arrows	infix	340	5	5	stretchy
&#x21F7;	⇷	leftwards arrow with vertical stroke	infix	340	5	5	stretchy
&#x21F8;	⇸	rightwards arrow with vertical stroke	infix	340	5	5	stretchy
&#x21F9;	⇹	left right arrow with vertical stroke	infix	340	5	5	stretchy
&#x21FA;	⇺	leftwards arrow with double vertical stroke	infix	340	5	5	stretchy
&#x21FB;	⇻	rightwards arrow with double vertical stroke	infix	340	5	5	stretchy
&#x21FC;	⇼	left right arrow with double vertical stroke	infix	340	5	5	stretchy
&#x21FD;	⇽	leftwards open-headed arrow	infix	340	5	5	stretchy
&#x21FE;	⇾	rightwards open-headed arrow	infix	340	5	5	stretchy
&#x21FF;	⇿	left right open-headed arrow	infix	340	5	5	stretchy
&#x2301;	⌁	electric arrow	infix	340	5	5	
&#x237C;	⍼	right angle with downwards zigzag arrow	infix	340	5	5	
&#x238B;	⎋	broken circle with northwest arrow	infix	340	5	5	
&#x2794;	➔	heavy wide-headed rightwards arrow	infix	340	5	5	stretchy
&#x2798;	➘	heavy south east arrow	infix	340	5	5	
&#x2799;	➙	heavy rightwards arrow	infix	340	5	5	stretchy
&#x279A;	➚	heavy north east arrow	infix	340	5	5	
&#x279B;	➛	drafting point rightwards arrow	infix	340	5	5	stretchy
&#x279C;	➜	heavy round-tipped rightwards arrow	infix	340	5	5	stretchy
&#x279D;	➝	triangle-headed rightwards arrow	infix	340	5	5	stretchy
&#x279E;	➞	heavy triangle-headed rightwards arrow	infix	340	5	5	stretchy
&#x279F;	➟	dashed triangle-headed rightwards arrow	infix	340	5	5	stretchy
&#x27A0;	➠	heavy dashed triangle-headed rightwards arrow	infix	340	5	5	stretchy
&#x27A1;	➡	black rightwards arrow	infix	340	5	5	stretchy
&#x27A5;	➥	heavy black curved downwards and rightwards arrow	infix	340	5	5	stretchy
&#x27A6;	➦	heavy black curved upwards and rightwards arrow	infix	340	5	5	stretchy
&#x27A7;	➧	squat black rightwards arrow	infix	340	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x27A8;	➨	heavy concave-pointed black rightwards arrow	infix	340	5	5	stretchy
&#x27A9;	➩	right-shaded white rightwards arrow	infix	340	5	5	stretchy
&#x27AA;	➪	left-shaded white rightwards arrow	infix	340	5	5	stretchy
&#x27AB;	➫	back-tilted shadowed white rightwards arrow	infix	340	5	5	stretchy
&#x27AC;	➬	front-tilted shadowed white rightwards arrow	infix	340	5	5	stretchy
&#x27AD;	➭	heavy lower right-shadowed white rightwards arrow	infix	340	5	5	stretchy
&#x27AE;	➮	heavy upper right-shadowed white rightwards arrow	infix	340	5	5	stretchy
&#x27AF;	➯	notched lower right-shadowed white rightwards arrow	infix	340	5	5	stretchy
&#x27B1;	➱	notched upper right-shadowed white rightwards arrow	infix	340	5	5	stretchy
&#x27B2;	➲	circled heavy white rightwards arrow	infix	340	5	5	
&#x27B3;	➳	white-feathered rightwards arrow	infix	340	5	5	stretchy
&#x27B4;	➴	black-feathered south east arrow	infix	340	5	5	
&#x27B5;	➵	black-feathered rightwards arrow	infix	340	5	5	stretchy
&#x27B6;	➶	black-feathered north east arrow	infix	340	5	5	
&#x27B7;	➷	heavy black-feathered south east arrow	infix	340	5	5	
&#x27B8;	➸	heavy black-feathered rightwards arrow	infix	340	5	5	stretchy
&#x27B9;	➹	heavy black-feathered north east arrow	infix	340	5	5	
&#x27BA;	➺	teardrop-barbed rightwards arrow	infix	340	5	5	stretchy
&#x27BB;	➻	heavy teardrop-shanked rightwards arrow	infix	340	5	5	stretchy
&#x27BC;	➼	wedge-tailed rightwards arrow	infix	340	5	5	stretchy
&#x27BD;	➽	heavy wedge-tailed rightwards arrow	infix	340	5	5	stretchy
&#x27BE;	➾	open-outlined rightwards arrow	infix	340	5	5	stretchy
&#x27F0;	⟰	upwards quadruple arrow	infix	340	5	5	stretchy
&#x27F1;	⟱	downwards quadruple arrow	infix	340	5	5	stretchy
&#x27F2;	⟲	anticlockwise gapped circle arrow	infix	340	5	5	
&#x27F3;	⟳	clockwise gapped circle arrow	infix	340	5	5	
&#x27F4;	⟴	right arrow with circled plus	infix	340	5	5	stretchy
&#x27F5;	⟵	long leftwards arrow	infix	340	5	5	stretchy
&#x27F6;	⟶	long rightwards arrow	infix	340	5	5	stretchy
&#x27F7;	⟷	long left right arrow	infix	340	5	5	stretchy
&#x27F8;	⟸	long leftwards double arrow	infix	340	5	5	stretchy
&#x27F9;	⟹	long rightwards double arrow	infix	340	5	5	stretchy
&#x27FA;	⟺	long left right double arrow	infix	340	5	5	stretchy
&#x27FB;	⟻	long leftwards arrow from bar	infix	340	5	5	stretchy
&#x27FC;	⟼	long rightwards arrow from bar	infix	340	5	5	stretchy
&#x27FD;	⟽	long leftwards double arrow from bar	infix	340	5	5	stretchy
&#x27FE;	⟾	long rightwards double arrow from bar	infix	340	5	5	stretchy
&#x27FF;	⟿	long rightwards squiggle arrow	infix	340	5	5	stretchy

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2900;	⤠	rightwards two-headed arrow with vertical stroke	infix	340	5	5	stretchy
&#x2901;	⤡	rightwards two-headed arrow with double vertical stroke	infix	340	5	5	stretchy
&#x2902;	⤢	leftwards double arrow with vertical stroke	infix	340	5	5	stretchy
&#x2903;	⤣	rightwards double arrow with vertical stroke	infix	340	5	5	stretchy
&#x2904;	⤤	left right double arrow with vertical stroke	infix	340	5	5	stretchy
&#x2905;	⤥	rightwards two-headed arrow from bar	infix	340	5	5	stretchy
&#x2906;	⤦	leftwards double arrow from bar	infix	340	5	5	stretchy
&#x2907;	⤧	rightwards double arrow from bar	infix	340	5	5	stretchy
&#x2908;	⤨	downwards arrow with horizontal stroke	infix	340	5	5	stretchy
&#x2909;	⤩	upwards arrow with horizontal stroke	infix	340	5	5	stretchy
&#x290A;	⤪	upwards triple arrow	infix	340	5	5	stretchy
&#x290B;	⤫	downwards triple arrow	infix	340	5	5	stretchy
&#x290C;	⤬	leftwards double dash arrow	infix	340	5	5	stretchy
&#x290D;	⤭	rightwards double dash arrow	infix	340	5	5	stretchy
&#x290E;	⤮	leftwards triple dash arrow	infix	340	5	5	stretchy
&#x290F;	⤯	rightwards triple dash arrow	infix	340	5	5	stretchy
&#x2910;	⤰	rightwards two-headed triple dash arrow	infix	340	5	5	stretchy
&#x2911;	⤱	rightwards arrow with dotted stem	infix	340	5	5	stretchy
&#x2912;	⤲	upwards arrow to bar	infix	340	5	5	stretchy
&#x2913;	⤳	downwards arrow to bar	infix	340	5	5	stretchy
&#x2914;	⤴	rightwards arrow with tail with vertical stroke	infix	340	5	5	stretchy
&#x2915;	⤵	rightwards arrow with tail with double vertical stroke	infix	340	5	5	stretchy
&#x2916;	⤶	rightwards two-headed arrow with tail	infix	340	5	5	stretchy
&#x2917;	⤷	rightwards two-headed arrow with tail with vertical stroke	infix	340	5	5	stretchy
&#x2918;	⤸	rightwards two-headed arrow with tail with double vertical stroke	infix	340	5	5	stretchy
&#x2919;	⤹	leftwards arrow-tail	infix	340	5	5	stretchy
&#x291A;	⤺	rightwards arrow-tail	infix	340	5	5	stretchy
&#x291B;	⤻	leftwards double arrow-tail	infix	340	5	5	stretchy
&#x291C;	⤼	rightwards double arrow-tail	infix	340	5	5	stretchy
&#x291D;	⤽	leftwards arrow to black diamond	infix	340	5	5	stretchy
&#x291E;	⤾	rightwards arrow to black diamond	infix	340	5	5	stretchy
&#x291F;	⤿	leftwards arrow from bar to black diamond	infix	340	5	5	stretchy
&#x2920;	⥀	rightwards arrow from bar to black diamond	infix	340	5	5	stretchy
&#x2921;	⥁	north west and south east arrow	infix	340	5	5	
&#x2922;	⥂	north east and south west arrow	infix	340	5	5	
&#x2923;	⥃	north west arrow with hook	infix	340	5	5	
&#x2924;	⥄	north east arrow with hook	infix	340	5	5	
&#x2925;	⥅	south east arrow with hook	infix	340	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2926;	⤦	south west arrow with hook	infix	340	5	5	
&#x2927;	⤧	north west arrow and north east arrow	infix	340	5	5	
&#x2928;	⤨	north east arrow and south east arrow	infix	340	5	5	
&#x2929;	⤩	south east arrow and south west arrow	infix	340	5	5	
&#x292A;	⤪	south west arrow and north west arrow	infix	340	5	5	
&#x292B;	⤫	rising diagonal crossing falling diagonal	infix	340	5	5	
&#x292C;	⤬	falling diagonal crossing rising diagonal	infix	340	5	5	
&#x292D;	⤭	south east arrow crossing north east arrow	infix	340	5	5	
&#x292E;	⤮	north east arrow crossing south east arrow	infix	340	5	5	
&#x292F;	⤯	falling diagonal crossing north east arrow	infix	340	5	5	
&#x2930;	⤰	rising diagonal crossing south east arrow	infix	340	5	5	
&#x2931;	⤱	north east arrow crossing north west arrow	infix	340	5	5	
&#x2932;	⤲	north west arrow crossing north east arrow	infix	340	5	5	
&#x2933;	⤳	wave arrow pointing directly right	infix	340	5	5	
&#x2934;	⤴	arrow pointing rightwards then curving upwards	infix	340	5	5	stretchy
&#x2935;	⤵	arrow pointing rightwards then curving downwards	infix	340	5	5	stretchy
&#x2936;	⤶	arrow pointing downwards then curving leftwards	infix	340	5	5	stretchy
&#x2937;	⤷	arrow pointing downwards then curving rightwards	infix	340	5	5	stretchy
&#x2938;	⤸	right-side arc clockwise arrow	infix	340	5	5	
&#x2939;	⤹	left-side arc anticlockwise arrow	infix	340	5	5	
&#x293A;	⤺	top arc anticlockwise arrow	infix	340	5	5	
&#x293B;	⤻	bottom arc anticlockwise arrow	infix	340	5	5	
&#x293C;	⤼	top arc clockwise arrow with minus	infix	340	5	5	
&#x293D;	⤽	top arc anticlockwise arrow with plus	infix	340	5	5	
&#x293E;	⤾	lower right semicircular clockwise arrow	infix	340	5	5	
&#x293F;	⤿	lower left semicircular anticlockwise arrow	infix	340	5	5	
&#x2940;	⥀	anticlockwise closed circle arrow	infix	340	5	5	
&#x2941;	⥁	clockwise closed circle arrow	infix	340	5	5	
&#x2942;	⥂	rightwards arrow above short leftwards arrow	infix	340	5	5	stretchy
&#x2943;	⥃	leftwards arrow above short rightwards arrow	infix	340	5	5	stretchy
&#x2944;	⥄	short rightwards arrow above leftwards arrow	infix	340	5	5	stretchy
&#x2945;	⥅	rightwards arrow with plus below	infix	340	5	5	stretchy
&#x2946;	⥆	leftwards arrow with plus below	infix	340	5	5	stretchy
&#x2947;	⥇	rightwards arrow through x	infix	340	5	5	stretchy
&#x2948;	⥈	left right arrow through small circle	infix	340	5	5	stretchy
&#x2949;	⥉	upwards two-headed arrow from small circle	infix	340	5	5	stretchy
&#x294A;	⥊	left barb up right barb down harpoon	infix	340	5	5	stretchy
&#x294B;	⥋	left barb down right barb up harpoon	infix	340	5	5	stretchy

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x294C;	⥌	up barb right down barb left harpoon	infix	340	5	5	stretchy
&#x294D;	⥍	up barb left down barb right harpoon	infix	340	5	5	stretchy
&#x294E;	⥎	left barb up right barb up harpoon	infix	340	5	5	stretchy
&#x294F;	⥏	up barb right down barb right harpoon	infix	340	5	5	stretchy
&#x2950;	⥐	left barb down right barb down harpoon	infix	340	5	5	stretchy
&#x2951;	⥑	up barb left down barb left harpoon	infix	340	5	5	stretchy
&#x2952;	⥒	leftwards harpoon with barb up to bar	infix	340	5	5	stretchy
&#x2953;	⥓	rightwards harpoon with barb up to bar	infix	340	5	5	stretchy
&#x2954;	⥔	upwards harpoon with barb right to bar	infix	340	5	5	stretchy
&#x2955;	⥕	downwards harpoon with barb right to bar	infix	340	5	5	stretchy
&#x2956;	⥖	leftwards harpoon with barb down to bar	infix	340	5	5	stretchy
&#x2957;	⥗	rightwards harpoon with barb down to bar	infix	340	5	5	stretchy
&#x2958;	⥘	upwards harpoon with barb left to bar	infix	340	5	5	stretchy
&#x2959;	⥙	downwards harpoon with barb left to bar	infix	340	5	5	stretchy
&#x295A;	⥚	leftwards harpoon with barb up from bar	infix	340	5	5	stretchy
&#x295B;	⥛	rightwards harpoon with barb up from bar	infix	340	5	5	stretchy
&#x295C;	⥜	upwards harpoon with barb right from bar	infix	340	5	5	stretchy
&#x295D;	⥝	downwards harpoon with barb right from bar	infix	340	5	5	stretchy
&#x295E;	⥞	leftwards harpoon with barb down from bar	infix	340	5	5	stretchy
&#x295F;	⥟	rightwards harpoon with barb down from bar	infix	340	5	5	stretchy
&#x2960;	⥠	upwards harpoon with barb left from bar	infix	340	5	5	stretchy
&#x2961;	⥡	downwards harpoon with barb left from bar	infix	340	5	5	stretchy
&#x2962;	⥢	leftwards harpoon with barb up above leftwards harpoon with barb down	infix	340	5	5	stretchy
&#x2963;	⥣	upwards harpoon with barb left beside upwards harpoon with barb right	infix	340	5	5	stretchy
&#x2964;	⥤	rightwards harpoon with barb up above rightwards harpoon with barb down	infix	340	5	5	stretchy
&#x2965;	⥥	downwards harpoon with barb left beside downwards harpoon with barb right	infix	340	5	5	stretchy
&#x2966;	⥦	leftwards harpoon with barb up above rightwards harpoon with barb up	infix	340	5	5	stretchy
&#x2967;	⥧	leftwards harpoon with barb down above rightwards harpoon with barb down	infix	340	5	5	stretchy
&#x2968;	⥨	rightwards harpoon with barb up above leftwards harpoon with barb up	infix	340	5	5	stretchy
&#x2969;	⥩	rightwards harpoon with barb down above leftwards harpoon with barb down	infix	340	5	5	stretchy
&#x296A;	⥪	leftwards harpoon with barb up above long dash	infix	340	5	5	stretchy
&#x296B;	⥫	leftwards harpoon with barb down below long dash	infix	340	5	5	stretchy
&#x296C;	⥬	rightwards harpoon with barb up above long dash	infix	340	5	5	stretchy
&#x296D;	⥭	rightwards harpoon with barb down below long dash	infix	340	5	5	stretchy
&#x296E;	⥮	upwards harpoon with barb left beside downwards harpoon with barb right	infix	340	5	5	stretchy

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x296F;	⥯	downwards harpoon with barb left beside upwards harpoon with barb right	infix	340	5	5	stretchy
&#x2970;	⥰	right double arrow with rounded head	infix	340	5	5	stretchy
&#x2971;	⥱	equals sign above rightwards arrow	infix	340	5	5	stretchy
&#x2972;	⥲	tilde operator above rightwards arrow	infix	340	5	5	stretchy
&#x2973;	⥳	leftwards arrow above tilde operator	infix	340	5	5	stretchy
&#x2974;	⥴	rightwards arrow above tilde operator	infix	340	5	5	stretchy
&#x2975;	⥵	rightwards arrow above almost equal to	infix	340	5	5	stretchy
&#x297C;	⥼	left fish tail	infix	340	5	5	stretchy
&#x297D;	⥽	right fish tail	infix	340	5	5	stretchy
&#x297E;	⥾	up fish tail	infix	340	5	5	stretchy
&#x297F;	⥿	down fish tail	infix	340	5	5	stretchy
&#x29DF;	⧟	double-ended multimap	infix	340	5	5	
&#x2B00;	⬀	north east white arrow	infix	340	5	5	
&#x2B01;	⬁	north west white arrow	infix	340	5	5	
&#x2B02;	⬂	south east white arrow	infix	340	5	5	
&#x2B03;	⬃	south west white arrow	infix	340	5	5	
&#x2B04;	⬄	left right white arrow	infix	340	5	5	stretchy
&#x2B05;	⬅	leftwards black arrow	infix	340	5	5	stretchy
&#x2B06;	⬆	upwards black arrow	infix	340	5	5	stretchy
&#x2B07;	⬇	downwards black arrow	infix	340	5	5	stretchy
&#x2B08;	⬈	north east black arrow	infix	340	5	5	
&#x2B09;	⬉	north west black arrow	infix	340	5	5	
&#x2B0A;	⬊	south east black arrow	infix	340	5	5	
&#x2B0B;	⬋	south west black arrow	infix	340	5	5	
&#x2B0C;	⬌	left right black arrow	infix	340	5	5	stretchy
&#x2B0D;	⬍	up down black arrow	infix	340	5	5	stretchy
&#x2B0E;	⬎	rightwards arrow with tip downwards	infix	340	5	5	stretchy
&#x2B0F;	⬏	rightwards arrow with tip upwards	infix	340	5	5	stretchy
&#x2B10;	⬐	leftwards arrow with tip downwards	infix	340	5	5	stretchy
&#x2B11;	⬑	leftwards arrow with tip upwards	infix	340	5	5	stretchy
&#x2B30;	⬰	left arrow with small circle	infix	340	5	5	stretchy
&#x2B31;	⬱	three leftwards arrows	infix	340	5	5	stretchy
&#x2B32;	⬲	left arrow with circled plus	infix	340	5	5	stretchy
&#x2B33;	⬳	long leftwards squiggle arrow	infix	340	5	5	stretchy
&#x2B34;	⬴	leftwards two-headed arrow with vertical stroke	infix	340	5	5	stretchy
&#x2B35;	⬵	leftwards two-headed arrow with double vertical stroke	infix	340	5	5	stretchy
&#x2B36;	⬶	leftwards two-headed arrow from bar	infix	340	5	5	stretchy
&#x2B37;	⬷	leftwards two-headed triple dash arrow	infix	340	5	5	stretchy
&#x2B38;	⬸	leftwards arrow with dotted stem	infix	340	5	5	stretchy

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2B39;	⬹	leftwards arrow with tail with vertical stroke	infix	340	5	5	stretchy
&#x2B3A;	⬺	leftwards arrow with tail with double vertical stroke	infix	340	5	5	stretchy
&#x2B3B;	⬻	leftwards two-headed arrow with tail	infix	340	5	5	stretchy
&#x2B3C;	⬼	leftwards two-headed arrow with tail with vertical stroke	infix	340	5	5	stretchy
&#x2B3D;	⬽	leftwards two-headed arrow with tail with double vertical stroke	infix	340	5	5	stretchy
&#x2B3E;	⬾	leftwards arrow through x	infix	340	5	5	stretchy
&#x2B3F;	⬿	wave arrow pointing directly left	infix	340	5	5	
&#x2B40;	⭀	equals sign above leftwards arrow	infix	340	5	5	stretchy
&#x2B41;	⭁	reverse tilde operator above leftwards arrow	infix	340	5	5	stretchy
&#x2B42;	⭂	leftwards arrow above reverse almost equal to	infix	340	5	5	stretchy
&#x2B43;	⭃	rightwards arrow through greater-than	infix	340	5	5	stretchy
&#x2B44;	⭄	rightwards arrow through superset	infix	340	5	5	stretchy
&#x2B45;	⭅	leftwards quadruple arrow	infix	340	5	5	stretchy
&#x2B46;	⭆	rightwards quadruple arrow	infix	340	5	5	stretchy
&#x2B47;	⭇	reverse tilde operator above rightwards arrow	infix	340	5	5	stretchy
&#x2B48;	⭈	rightwards arrow above reverse almost equal to	infix	340	5	5	stretchy
&#x2B49;	⭉	tilde operator above leftwards arrow	infix	340	5	5	stretchy
&#x2B4A;	⭊	leftwards arrow above almost equal to	infix	340	5	5	stretchy
&#x2B4B;	⭋	leftwards arrow above reverse tilde operator	infix	340	5	5	stretchy
&#x2B4C;	⭌	rightwards arrow above reverse tilde operator	infix	340	5	5	stretchy
&#x2B4D;	⭍	downwards triangle-headed zigzag arrow	infix	340	5	5	
&#x2B4E;	⭎	short slanted north arrow	infix	340	5	5	
&#x2B4F;	⭏	short backslanted south arrow	infix	340	5	5	
&#x2B5A;	⭚	slanted north arrow with hooked head	infix	340	5	5	
&#x2B5B;	⭛	backslanted south arrow with hooked tail	infix	340	5	5	
&#x2B5C;	⭜	slanted north arrow with horizontal tail	infix	340	5	5	
&#x2B5D;	⭝	backslanted south arrow with horizontal tail	infix	340	5	5	
&#x2B5E;	⭞	bent arrow pointing downwards then north east	infix	340	5	5	
&#x2B5F;	⭟	short bent arrow pointing downwards then north east	infix	340	5	5	
&#x2B60;	⭠	leftwards triangle-headed arrow	infix	340	5	5	stretchy
&#x2B61;	⭡	upwards triangle-headed arrow	infix	340	5	5	stretchy
&#x2B62;	⭢	rightwards triangle-headed arrow	infix	340	5	5	stretchy
&#x2B63;	⭣	downwards triangle-headed arrow	infix	340	5	5	stretchy
&#x2B64;	⭤	left right triangle-headed arrow	infix	340	5	5	stretchy
&#x2B65;	⭥	up down triangle-headed arrow	infix	340	5	5	stretchy
&#x2B66;	⭦	north west triangle-headed arrow	infix	340	5	5	
&#x2B67;	⭧	north east triangle-headed arrow	infix	340	5	5	
&#x2B68;	⭨	south east triangle-headed arrow	infix	340	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2B69;		south west triangle-headed arrow	infix	340	5	5	
&#x2B6A;		leftwards triangle-headed dashed arrow	infix	340	5	5	stretchy
&#x2B6B;		upwards triangle-headed dashed arrow	infix	340	5	5	stretchy
&#x2B6C;		rightwards triangle-headed dashed arrow	infix	340	5	5	stretchy
&#x2B6D;		downwards triangle-headed dashed arrow	infix	340	5	5	stretchy
&#x2B6E;		clockwise triangle-headed open circle arrow	infix	340	5	5	
&#x2B6F;		anticlockwise triangle-headed open circle arrow	infix	340	5	5	
&#x2B70;		leftwards triangle-headed arrow to bar	infix	340	5	5	stretchy
&#x2B71;		upwards triangle-headed arrow to bar	infix	340	5	5	stretchy
&#x2B72;		rightwards triangle-headed arrow to bar	infix	340	5	5	stretchy
&#x2B73;		downwards triangle-headed arrow to bar	infix	340	5	5	stretchy
&#x2B76;		north west triangle-headed arrow to bar	infix	340	5	5	
&#x2B77;		north east triangle-headed arrow to bar	infix	340	5	5	
&#x2B78;		south east triangle-headed arrow to bar	infix	340	5	5	
&#x2B79;		south west triangle-headed arrow to bar	infix	340	5	5	
&#x2B7A;		leftwards triangle-headed arrow with double horizontal stroke	infix	340	5	5	stretchy
&#x2B7B;		upwards triangle-headed arrow with double horizontal stroke	infix	340	5	5	stretchy
&#x2B7C;		rightwards triangle-headed arrow with double horizontal stroke	infix	340	5	5	stretchy
&#x2B7D;		downwards triangle-headed arrow with double horizontal stroke	infix	340	5	5	stretchy
&#x2B80;		leftwards triangle-headed arrow over rightwards triangle-headed arrow	infix	340	5	5	stretchy
&#x2B81;		upwards triangle-headed arrow leftwards of downwards triangle-headed arrow	infix	340	5	5	stretchy
&#x2B82;	⇄	rightwards triangle-headed arrow over leftwards triangle-headed arrow	infix	340	5	5	stretchy
&#x2B83;	⇅	downwards triangle-headed arrow leftwards of upwards triangle-headed arrow	infix	340	5	5	stretchy
&#x2B84;		leftwards triangle-headed paired arrows	infix	340	5	5	stretchy
&#x2B85;		upwards triangle-headed paired arrows	infix	340	5	5	stretchy
&#x2B86;		rightwards triangle-headed paired arrows	infix	340	5	5	stretchy
&#x2B87;		downwards triangle-headed paired arrows	infix	340	5	5	stretchy
&#x2B88;		leftwards black circled white arrow	infix	340	5	5	
&#x2B89;		upwards black circled white arrow	infix	340	5	5	
&#x2B8A;		rightwards black circled white arrow	infix	340	5	5	
&#x2B8B;		downwards black circled white arrow	infix	340	5	5	
&#x2B8C;		anticlockwise triangle-headed right u-shaped arrow	infix	340	5	5	
&#x2B8D;		anticlockwise triangle-headed bottom u-shaped arrow	infix	340	5	5	
&#x2B8E;		anticlockwise triangle-headed left u-shaped arrow	infix	340	5	5	
&#x2B8F;		anticlockwise triangle-headed top u-shaped arrow	infix	340	5	5	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2B94;	⭔	four corner arrows circling anticlockwise	infix	340	5	5	
&#x2B95;	➡	rightwards black arrow	infix	340	5	5	stretchy
&#x2BA0;	⮠	downwards triangle-headed arrow with long tip leftwards	infix	340	5	5	stretchy
&#x2BA1;	⮡	downwards triangle-headed arrow with long tip rightwards	infix	340	5	5	stretchy
&#x2BA2;	⮢	upwards triangle-headed arrow with long tip leftwards	infix	340	5	5	stretchy
&#x2BA3;	⮣	upwards triangle-headed arrow with long tip rightwards	infix	340	5	5	stretchy
&#x2BA4;	⮤	leftwards triangle-headed arrow with long tip upwards	infix	340	5	5	stretchy
&#x2BA5;	⮥	rightwards triangle-headed arrow with long tip upwards	infix	340	5	5	stretchy
&#x2BA6;	⮦	leftwards triangle-headed arrow with long tip downwards	infix	340	5	5	stretchy
&#x2BA7;	⮧	rightwards triangle-headed arrow with long tip downwards	infix	340	5	5	stretchy
&#x2BA8;	⮨	black curved downwards and leftwards arrow	infix	340	5	5	stretchy
&#x2BA9;	⮩	black curved downwards and rightwards arrow	infix	340	5	5	stretchy
&#x2BAA;	⮪	black curved upwards and leftwards arrow	infix	340	5	5	stretchy
&#x2BAB;	⮫	black curved upwards and rightwards arrow	infix	340	5	5	stretchy
&#x2BAC;	⮬	black curved leftwards and upwards arrow	infix	340	5	5	stretchy
&#x2BAD;	⮭	black curved rightwards and upwards arrow	infix	340	5	5	stretchy
&#x2BAE;	⮮	black curved leftwards and downwards arrow	infix	340	5	5	stretchy
&#x2BAF;	⮯	black curved rightwards and downwards arrow	infix	340	5	5	stretchy
&#x2BB0;	⮰	ribbon arrow down left	infix	340	5	5	
&#x2BB1;	⮱	ribbon arrow down right	infix	340	5	5	
&#x2BB2;	⮲	ribbon arrow up left	infix	340	5	5	
&#x2BB3;	⮳	ribbon arrow up right	infix	340	5	5	
&#x2BB4;	⮴	ribbon arrow left up	infix	340	5	5	
&#x2BB5;	⮵	ribbon arrow right up	infix	340	5	5	
&#x2BB6;	⮶	ribbon arrow left down	infix	340	5	5	
&#x2BB7;	⮷	ribbon arrow right down	infix	340	5	5	
&#x2BB8;	⮸	upwards white arrow from bar with horizontal bar	infix	340	5	5	stretchy
&#x222A;	∪	union	infix	360	4	4	
&#x228C;	⊌	multiset	infix	360	4	4	
&#x228D;	⊍	multiset multiplication	infix	360	4	4	
&#x228E;	⊎	multiset union	infix	360	4	4	
&#x2294;	⊔	square cup	infix	360	4	4	
&#x22D3;	⋓	double union	infix	360	4	4	
&#x2A41;	⩁	union with minus sign	infix	360	4	4	
&#x2A42;	⩂	union with overbar	infix	360	4	4	
&#x2A45;	⩅	union with logical or	infix	360	4	4	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2A4A;	⩊	union beside and joined with union	infix	360	4	4	
&#x2A4C;	⩌	closed union with serifs	infix	360	4	4	
&#x2A4F;	⩏	double square union	infix	360	4	4	
&#x2229;	∩	intersection	infix	380	4	4	
&#x2293;	⊓	square cap	infix	380	4	4	
&#x22D2;	⋒	double intersection	infix	380	4	4	
&#x2A1F;	⨟	z notation schema composition	infix	380	4	4	
&#x2A20;	⨠	z notation schema piping	infix	380	4	4	
&#x2A21;	⨡	z notation schema projection	infix	380	4	4	
&#x2A3E;	⨾	z notation relational composition	infix	380	4	4	
&#x2A40;	⩀	intersection with dot	infix	380	4	4	
&#x2A43;	⩃	intersection with overbar	infix	380	4	4	
&#x2A44;	⩄	intersection with logical and	infix	380	4	4	
&#x2A46;	⩆	union above intersection	infix	380	4	4	
&#x2A47;	⩇	intersection above union	infix	380	4	4	
&#x2A48;	⩈	union above bar above intersection	infix	380	4	4	
&#x2A49;	⩉	intersection above bar above union	infix	380	4	4	
&#x2A4B;	⩋	intersection beside and joined with intersection	infix	380	4	4	
&#x2A4D;	⩍	closed intersection with serifs	infix	380	4	4	
&#x2A4E;	⩎	double square intersection	infix	380	4	4	
&#x2ADB;	⫛	transversal intersection	infix	380	4	4	
+	+	plus sign	infix	400	4	4	
-	-	hyphen-minus	infix	400	4	4	
&#xB1;	±	plus-minus sign	infix	400	4	4	
&#x2212;	−	minus sign	infix	400	4	4	
&#x2213;	∓	minus-or-plus sign	infix	400	4	4	
&#x2214;	∔	dot plus	infix	400	4	4	
&#x2216;	\	set minus	infix	400	4	4	
&#x2228;	∨	logical or	infix	400	4	4	
&#x2238;	∸	dot minus	infix	400	4	4	
&#x2295;	⊕	circled plus	infix	400	4	4	
&#x2296;	⊖	circled minus	infix	400	4	4	
&#x229D;	⊝	circled dash	infix	400	4	4	
&#x229E;	⊞	squared plus	infix	400	4	4	
&#x229F;	⊟	squared minus	infix	400	4	4	
&#x22BD;	⊽	nor	infix	400	4	4	
&#x22CE;	⋎	curly logical or	infix	400	4	4	
&#x2795;	➕	heavy plus sign	infix	400	4	4	
&#x2796;	➖	heavy minus sign	infix	400	4	4	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x29B8;	⦸	circled reverse solidus	infix	400	4	4	
&#x29C5;	⧅	squared falling diagonal slash	infix	400	4	4	
&#x29F5;	⧵	reverse solidus operator	infix	400	4	4	
&#x29F7;	⧷	reverse solidus with horizontal stroke	infix	400	4	4	
&#x29F9;	⧹	big reverse solidus	infix	400	4	4	
&#x29FA;	⧺	double plus	infix	400	4	4	
&#x29FB;	⧻	triple plus	infix	400	4	4	
&#x2A22;	⨢	plus sign with small circle above	infix	400	4	4	
&#x2A23;	⨣	plus sign with circumflex accent above	infix	400	4	4	
&#x2A24;	⨤	plus sign with tilde above	infix	400	4	4	
&#x2A25;	⨥	plus sign with dot below	infix	400	4	4	
&#x2A26;	⨦	plus sign with tilde below	infix	400	4	4	
&#x2A27;	⨧	plus sign with subscript two	infix	400	4	4	
&#x2A28;	⨨	plus sign with black triangle	infix	400	4	4	
&#x2A29;	⨩	minus sign with comma above	infix	400	4	4	
&#x2A2A;	⨪	minus sign with dot below	infix	400	4	4	
&#x2A2B;	⨫	minus sign with falling dots	infix	400	4	4	
&#x2A2C;	⨬	minus sign with rising dots	infix	400	4	4	
&#x2A2D;	⨭	plus sign in left half circle	infix	400	4	4	
&#x2A2E;	⨮	plus sign in right half circle	infix	400	4	4	
&#x2A39;	⨹	plus sign in triangle	infix	400	4	4	
&#x2A3A;	⨺	minus sign in triangle	infix	400	4	4	
&#x2A52;	⩒	logical or with dot above	infix	400	4	4	
&#x2A54;	⩔	double logical or	infix	400	4	4	
&#x2A56;	⩖	two intersecting logical or	infix	400	4	4	
&#x2A57;	⩗	sloping large or	infix	400	4	4	
&#x2A5B;	⩛	logical or with middle stem	infix	400	4	4	
&#x2A5D;	⩝	logical or with horizontal dash	infix	400	4	4	
&#x2A61;	⩡	small vee with underbar	infix	400	4	4	
&#x2A62;	⩢	logical or with double overbar	infix	400	4	4	
&#x2A63;	⩣	logical or with double underbar	infix	400	4	4	
&#x22BB;	⊻	xor	infix	420	4	4	
&#x2211;	∑	n-ary summation	prefix	440	3	3	largeop, movablelimits, symmetric
&#x2A0A;	⨊	modulo two sum	prefix	440	3	3	largeop, movablelimits, symmetric
&#x2A0B;	⨋	summation with integral	prefix	440	3	3	largeop, symmetric
&#x2A1D;	⨝	join	prefix	440	3	3	largeop, movablelimits, symmetric

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2A1E;	⨞	large left triangle operator	prefix	440	3	3	largeop, movablelimits, symmetric
&#x2A01;	⨁	n-ary circled plus operator	prefix	460	3	3	largeop, movablelimits, symmetric
&#x222B;	∫	integral	prefix	480	3	3	largeop, symmetric
&#x222C;	∬	double integral	prefix	480	3	3	largeop, symmetric
&#x222D;	∭	triple integral	prefix	480	3	3	largeop, symmetric
&#x222E;	∮	contour integral	prefix	480	3	3	largeop, symmetric
&#x222F;	∯	surface integral	prefix	480	3	3	largeop, symmetric
&#x2230;	∰	volume integral	prefix	480	3	3	largeop, symmetric
&#x2231;	∱	clockwise integral	prefix	480	3	3	largeop, symmetric
&#x2232;	∲	clockwise contour integral	prefix	480	3	3	largeop, symmetric
&#x2233;	∳	anticlockwise contour integral	prefix	480	3	3	largeop, symmetric
&#x2A0C;	⨌	quadruple integral operator	prefix	480	3	3	largeop, symmetric
&#x2A0D;	⨍	finite part integral	prefix	480	3	3	largeop, symmetric
&#x2A0E;	⨎	integral with double stroke	prefix	480	3	3	largeop, symmetric
&#x2A0F;	⨏	integral average with slash	prefix	480	3	3	largeop, symmetric
&#x2A10;	⨐	circulation function	prefix	480	3	3	largeop, symmetric
&#x2A11;	⨑	anticlockwise integration	prefix	480	3	3	largeop, symmetric
&#x2A12;	⨒	line integration with rectangular path around pole	prefix	480	3	3	largeop, symmetric
&#x2A13;	⨓	line integration with semicircular path around pole	prefix	480	3	3	largeop, symmetric
&#x2A14;	⨔	line integration not including the pole	prefix	480	3	3	largeop, symmetric
&#x2A15;	⨕	integral around a point operator	prefix	480	3	3	largeop, symmetric
&#x2A16;	⨖	quaternion integral operator	prefix	480	3	3	largeop, symmetric
&#x2A17;	⨗	integral with leftwards arrow with hook	prefix	480	3	3	largeop, symmetric
&#x2A18;	⨘	integral with times sign	prefix	480	3	3	largeop, symmetric
&#x2A19;	⨙	integral with intersection	prefix	480	3	3	largeop, symmetric
&#x2A1A;	⨚	integral with union	prefix	480	3	3	largeop, symmetric
&#x2A1B;	⨛	integral with overbar	prefix	480	3	3	largeop, symmetric
&#x2A1C;	⨜	integral with underbar	prefix	480	3	3	largeop, symmetric
&#x22C3;	⋃	n-ary union	prefix	500	3	3	largeop, movablelimits, symmetric
&#x2A03;	⨃	n-ary union operator with dot	prefix	500	3	3	largeop, movablelimits, symmetric
&#x2A04;	⨄	n-ary union operator with plus	prefix	500	3	3	largeop, movablelimits, symmetric
&#x22C0;	⋀	n-ary logical and	prefix	520	3	3	largeop, movablelimits, symmetric
&#x22C1;	⋁	n-ary logical or	prefix	520	3	3	largeop, movablelimits, symmetric

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x22C2;	⋂	n-ary intersection	prefix	520	3	3	largeop, movablelimits, symmetric
&#x2A00;	⨀	n-ary circled dot operator	prefix	520	3	3	largeop, movablelimits, symmetric
&#x2A02;	⨂	n-ary circled times operator	prefix	520	3	3	largeop, movablelimits, symmetric
&#x2A05;	⨅	n-ary square intersection operator	prefix	520	3	3	largeop, movablelimits, symmetric
&#x2A06;	⨆	n-ary square union operator	prefix	520	3	3	largeop, movablelimits, symmetric
&#x2A07;	⩇	two logical and operator	prefix	520	3	3	largeop, movablelimits, symmetric
&#x2A08;	⩈	two logical or operator	prefix	520	3	3	largeop, movablelimits, symmetric
&#x2A09;	⨉	n-ary times operator	prefix	520	3	3	largeop, movablelimits, symmetric
&#x2AFC;	⫼	large triple vertical bar operator	prefix	520	3	3	largeop, movablelimits, symmetric
&#x2AFF;	⫿	n-ary white vertical bar	prefix	520	3	3	largeop, movablelimits, symmetric
&#x220F;	∏	n-ary product	prefix	540	3	3	largeop, movablelimits, symmetric
&#x2210;	∐	n-ary coproduct	prefix	540	3	3	largeop, movablelimits, symmetric
@	@	commercial at	infix	560	3	3	
&#x221F;	∟	right angle	prefix	580	0	0	
&#x2220;	∠	angle	prefix	580	0	0	
&#x2221;	∡	measured angle	prefix	580	0	0	
&#x2222;	∢	spherical angle	prefix	580	0	0	
&#x22BE;	⊾	right angle with arc	prefix	580	0	0	
&#x22BF;	⊿	right triangle	prefix	580	0	0	
&#x27C0;	⟀	three dimensional angle	prefix	580	0	0	
&#x299B;	⦛	measured angle opening left	prefix	580	0	0	
&#x299C;	⦜	right angle variant with square	prefix	580	0	0	
&#x299D;	⦝	measured right angle with dot	prefix	580	0	0	
&#x299E;	⦞	angle with s inside	prefix	580	0	0	
&#x299F;	⦟	acute angle	prefix	580	0	0	
&#x29A0;	⦠	spherical angle opening left	prefix	580	0	0	
&#x29A1;	⦡	spherical angle opening up	prefix	580	0	0	
&#x29A2;	⦢	turned angle	prefix	580	0	0	
&#x29A3;	⦣	reversed angle	prefix	580	0	0	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x29A4;	⦤	angle with underbar	prefix	580	0	0	
&#x29A5;	⦥	reversed angle with underbar	prefix	580	0	0	
&#x29A6;	⦦	oblique angle opening up	prefix	580	0	0	
&#x29A7;	⦧	oblique angle opening down	prefix	580	0	0	
&#x29A8;	⦨	measured angle with open arm ending in arrow pointing up and right	prefix	580	0	0	
&#x29A9;	⦩	measured angle with open arm ending in arrow pointing up and left	prefix	580	0	0	
&#x29AA;	⦪	measured angle with open arm ending in arrow pointing down and right	prefix	580	0	0	
&#x29AB;	⦫	measured angle with open arm ending in arrow pointing down and left	prefix	580	0	0	
&#x29AC;	⦬	measured angle with open arm ending in arrow pointing right and up	prefix	580	0	0	
&#x29AD;	⦭	measured angle with open arm ending in arrow pointing left and up	prefix	580	0	0	
&#x29AE;	⦮	measured angle with open arm ending in arrow pointing right and down	prefix	580	0	0	
&#x29AF;	⦯	measured angle with open arm ending in arrow pointing left and down	prefix	580	0	0	
&&	&&	multiple character operator: &&	infix	600	4	4	
&#x2227;	∧	logical and	infix	600	4	4	
&#x22BC;	⊼	nand	infix	600	4	4	
&#x22CF;	⋏	curly logical and	infix	600	4	4	
&#x2A51;	⩑	logical and with dot above	infix	600	4	4	
&#x2A53;	⩓	double logical and	infix	600	4	4	
&#x2A55;	⩕	two intersecting logical and	infix	600	4	4	
&#x2A58;	⩘	sloping large and	infix	600	4	4	
&#x2A59;	⩙	logical or overlapping logical and	infix	600	4	4	
&#x2A5A;	⩚	logical and with middle stem	infix	600	4	4	
&#x2A5C;	⩜	logical and with horizontal dash	infix	600	4	4	
&#x2A5E;	⩞	logical and with double overbar	infix	600	4	4	
&#x2A5F;	⩟	logical and with underbar	infix	600	4	4	
&#x2A60;	⩠	logical and with double underbar	infix	600	4	4	
*	*	asterisk	infix	620	3	3	
.	.	full stop	infix	620	3	3	
&#xB7;	·	middle dot	infix	620	3	3	
&#xD7;	×	multiplication sign	infix	620	3	3	
&#x2022;	•	bullet	infix	620	3	3	
&#x2043;	⁃	hyphen bullet	infix	620	3	3	
&#x2062;		invisible times	infix	620	0	0	
&#x2217;	∗	asterisk operator	infix	620	3	3	
&#x2219;	∙	bullet operator	infix	620	3	3	
&#x2240;	≀	wreath product	infix	620	3	3	
&#x2297;	⊗	circled times	infix	620	3	3	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x2299;	⊙	circled dot operator	infix	620	3	3	
&#x229B;	⊛	circled asterisk operator	infix	620	3	3	
&#x22A0;	⊠	squared times	infix	620	3	3	
&#x22A1;	⊡	squared dot operator	infix	620	3	3	
&#x22BA;	⊺	intercalate	infix	620	3	3	
&#x22C5;	⋅	dot operator	infix	620	3	3	
&#x22C6;	⋆	star operator	infix	620	3	3	
&#x22C7;	⋇	division times	infix	620	3	3	
&#x22C9;	⋉	left normal factor semidirect product	infix	620	3	3	
&#x22CA;	⋊	right normal factor semidirect product	infix	620	3	3	
&#x22CB;	⋋	left semidirect product	infix	620	3	3	
&#x22CC;	⋌	right semidirect product	infix	620	3	3	
&#x2305;	⌅	projective	infix	620	3	3	
&#x2306;	⌆	perspective	infix	620	3	3	
&#x29C6;	⧆	squared asterisk	infix	620	3	3	
&#x29C8;	⧈	squared square	infix	620	3	3	
&#x29D4;	⧔	times with left half black	infix	620	3	3	
&#x29D5;	⧕	times with right half black	infix	620	3	3	
&#x29D6;	⧖	white hourglass	infix	620	3	3	
&#x29D7;	⧗	black hourglass	infix	620	3	3	
&#x29E2;	⧢	shuffle product	infix	620	3	3	
&#x2A1D;	⨝	join	infix	620	3	3	
&#x2A1E;	⨞	large left triangle operator	infix	620	3	3	
&#x2A2F;	×	vector or cross product	infix	620	3	3	
&#x2A30;	⨰	multiplication sign with dot above	infix	620	3	3	
&#x2A31;	⨱	multiplication sign with underbar	infix	620	3	3	
&#x2A32;	⨲	semidirect product with bottom closed	infix	620	3	3	
&#x2A33;	⨳	smash product	infix	620	3	3	
&#x2A34;	⨴	multiplication sign in left half circle	infix	620	3	3	
&#x2A35;	⨵	multiplication sign in right half circle	infix	620	3	3	
&#x2A36;	⨶	circled multiplication sign with circumflex accent	infix	620	3	3	
&#x2A37;	⨷	multiplication sign in double circle	infix	620	3	3	
&#x2A3B;	⨻	multiplication sign in triangle	infix	620	3	3	
&#x2A3C;	⨼	interior product	infix	620	3	3	
&#x2A3D;	⨽	righthand interior product	infix	620	3	3	
&#x2A3F;	⨿	amalgamation or coproduct	infix	620	3	3	
&#x2A50;	⩐	closed union with serifs and smash product	infix	620	3	3	
%	%	percent sign	infix	640	3	3	
\	\	reverse solidus	infix	660	0	0	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
/	/	solidus	infix	680	4	4	
÷	÷	division sign	infix	680	4	4	
⁄	⁄	fraction slash	infix	680	4	4	
∕	∕	division slash	infix	680	4	4	
∶	∶	ratio	infix	680	4	4	
⊘	⊘	circled division slash	infix	680	4	4	
➗	➗	heavy division sign	infix	680	4	4	
⟋	⟋	mathematical rising diagonal	infix	680	3	3	
⟍	⟍	mathematical falling diagonal	infix	680	3	3	
⦼	⦼	circled anticlockwise-rotated division sign	infix	680	4	4	
⧄	⧄	squared rising diagonal slash	infix	680	4	4	
⧶	⧶	solidus with overbar	infix	680	4	4	
⧸	⧸	big solidus	infix	680	4	4	
⨸	⨸	circled division sign	infix	680	4	4	
⫶	⫶	triple colon operator	infix	680	4	4	
⫻	⫻	triple solidus binary relation	infix	680	4	4	
⫽	⫽	double solidus operator	infix	680	4	4	
⫾	⫾	white vertical bar	infix	680	3	3	
⩤	⩤	z notation domain antirestriction	infix	700	3	3	
⩥	⩥	z notation range antirestriction	infix	700	3	3	
+	+	plus sign	prefix	720	0	0	
-	-	hyphen-minus	prefix	720	0	0	
±	±	plus-minus sign	prefix	720	0	0	
∁	∁	complement	prefix	720	0	0	
∆	∆	increment	infix	720	0	0	
−	−	minus sign	prefix	720	0	0	
∓	∓	minus-or-plus sign	prefix	720	0	0	
➕	➕	heavy plus sign	prefix	720	0	0	
➖	➖	heavy minus sign	prefix	720	0	0	
⫝̸	⫝̸	forking	infix	740	3	3	
⫝	⫝	nonforking	infix	740	3	3	
**	**	multiple character operator: **	infix	760	3	3	
ⅅ	ⅅ	double-struck italic capital d	prefix	780	3	0	
ⅆ	ⅆ	double-struck italic small d	prefix	780	3	0	
∂	∂	partial differential	prefix	780	3	0	
∇	∇	nabla	prefix	780	0	0	
<>	<>	multiple character operator: <>	infix	800	3	3	
^	^	circumflex accent	infix	800	3	3	
!	!	exclamation mark	postfix	820	0	0	
!!	!!	multiple character operator: !!	postfix	820	0	0	

Character	Glyph	Name	form	priority	lspace	rspace	Properties
%	%	percent sign	postfix	820	0	0	
′	′	prime	postfix	820	0	0	
?	?	question mark	infix	840	3	3	
√	√	square root	prefix	860	3	0	
∛	∛	cube root	prefix	860	3	0	
∜	∜	fourth root	prefix	860	3	0	
⁡		function application	infix	880	0	0	
∘	∘	ring operator	infix	900	3	3	
⊚	⊚	circled ring operator	infix	900	3	3	
⋄	⋄	diamond operator	infix	900	3	3	
⧇	⧇	squared small circle	infix	900	3	3	
"	"	quotation mark	postfix	920	0	0	
&amp;	&	ampersand	postfix	920	0	0	
'	'	apostrophe	postfix	920	0	0	
++	++	multiple character operator: ++	postfix	920	0	0	
--	--	multiple character operator: --	postfix	920	0	0	
^	^	circumflex accent	postfix	920	0	0	stretchy
_	_	low line	postfix	920	0	0	stretchy
`	`	grave accent	postfix	920	0	0	
~	~	tilde	postfix	920	0	0	stretchy
¨	¨	diaeresis	postfix	920	0	0	
¯	¯	macron	postfix	920	0	0	stretchy
°	°	degree sign	postfix	920	0	0	
²	2	superscript two	postfix	920	0	0	
³	3	superscript three	postfix	920	0	0	
´	´	acute accent	postfix	920	0	0	
¸	¸	cedilla	postfix	920	0	0	
¹	1	superscript one	postfix	920	0	0	
ˆ	ˆ	modifier letter circumflex accent	postfix	920	0	0	stretchy
ˇ	ˇ	caron	postfix	920	0	0	stretchy
ˉ	¯	modifier letter macron	postfix	920	0	0	stretchy
ˊ	´	modifier letter acute accent	postfix	920	0	0	
ˋ	`	modifier letter grave accent	postfix	920	0	0	
ˍ	_	modifier letter low macron	postfix	920	0	0	stretchy
˘	˘	breve	postfix	920	0	0	
˙	˙	dot above	postfix	920	0	0	
˚	°	ring above	postfix	920	0	0	
˜	˜	small tilde	postfix	920	0	0	stretchy
˝	˝	double acute accent	postfix	920	0	0	
˷	˷	modifier letter low tilde	postfix	920	0	0	stretchy

Character	Glyph	Name	form	priority	lspace	rspace	Properties
&#x302;	ˆ	combining circumflex accent	postfix	920	0	0	stretchy
&#x311;	̑	combining inverted breve	postfix	920	0	0	
&#x201A;	‚	single low-9 quotation mark	postfix	920	0	0	
&#x201B;	‛	single high-reversed-9 quotation mark	postfix	920	0	0	
&#x201E;	„	double low-9 quotation mark	postfix	920	0	0	
&#x201F;	‟	double high-reversed-9 quotation mark	postfix	920	0	0	
&#x2033;	″	double prime	postfix	920	0	0	
&#x2034;	‴	triple prime	postfix	920	0	0	
&#x2035;	‵	reversed prime	postfix	920	0	0	
&#x2036;	‶	reversed double prime	postfix	920	0	0	
&#x2037;	‷	reversed triple prime	postfix	920	0	0	
&#x203E;	‾	overline	postfix	920	0	0	stretchy
&#x2057;	⁗	quadruple prime	postfix	920	0	0	
&#x2064;		invisible plus	infix	920	0	0	
&#x20DB;	⃛	combining three dots above	postfix	920	0	0	
&#x20DC;	⃜	combining four dots above	postfix	920	0	0	
&#x2322;	⌢	frown	postfix	920	0	0	stretchy
&#x2323;	⌣	smile	postfix	920	0	0	stretchy
&#x23B4;	⎴	top square bracket	postfix	920	0	0	stretchy
&#x23B5;	⎵	bottom square bracket	postfix	920	0	0	stretchy
&#x23CD;	⏍	square foot	postfix	920	0	0	
&#x23DC;	⏜	top parenthesis	postfix	920	0	0	stretchy
&#x23DD;	⏝	bottom parenthesis	postfix	920	0	0	stretchy
&#x23DE;	⏞	top curly bracket	postfix	920	0	0	stretchy
&#x23DF;	⏟	bottom curly bracket	postfix	920	0	0	stretchy
&#x23E0;	⏠	top tortoise shell bracket	postfix	920	0	0	stretchy
&#x23E1;	⏡	bottom tortoise shell bracket	postfix	920	0	0	stretchy
&#x1EEF0;	ﷺ	arabic mathematical operator meem with hah with tatweel	postfix	920	0	0	stretchy
&#x1EEF1;	ﷻ	arabic mathematical operator hah with dal	postfix	920	0	0	stretchy
_	_	low line	infix	940	0	0	

C. MathML Accessibility

C.1 Introduction

As an essential element of the Open Web Platform, the W3C MathML specification has the unprecedented potential to enable content authors and developers to incorporate mathematical expressions on the web in such a way that the underlying structural and semantic information can be exposed to other technologies. Enabling this information exposure is

foundational for accessibility, as well as providing a path for making digital mathematics content machine readable, searchable and reusable

The internationally accepted standards and underpinning principles for creating accessible digital content on the web can be found in the W3C's Web Content Accessibility Guidelines [WCAG21]. In extending these principles to digital content containing mathematical information, WCAG provides a useful framework for defining accessibility wherever MathML is used.

As the current WCAG guidelines provide no direct guidance on how to ensure mathematical content encoded as MathML will be accessible to users with disabilities, this specification defines how to apply these guidelines to digital content containing MathML.

A benefit of following these recommendations is that it helps to ensure that digital mathematics content meets the accessibility requirements already widely used around the world for web content. In addition, ensuring that digital mathematics materials are accessible will expand the readership of such content to both readers with and without disabilities.

Additional guidance on best practices will be developed over time in [MathML-Notes]. Placing these in Notes allows them to adapt and evolve independent of the MathML specification, since accessibility practices often need more frequent updating. The Notes are also intended for use with past, present, and future versions of MathML, in addition to considerations for both the MathML-Core and the full MathML specification. The approach of a separate document ensures that the evolution of MathML does not lock accessibility best practices in time, and allows content authors to apply the most recent accessibility practices.

C.2 Accessibility benefits of using MathML

Many of the advances of mathematics in the modern world (i.e., since the late Renaissance) were arguably aided by the development of early symbolic notation which continues to be evolved in our present day. While simple literacy text can be used to state underlying mathematical concepts, symbolic notation provides a succinct method of representing abstract mathematical constructs in a portable manner which can be more easily consumed, manipulated and understood by humans and machines. Mathematics notation is itself a language intended for more than just visual rendering, inspection and manipulation, as it is also intended to express the underlying meaning of the author. These characteristics of mathematical notation have in turn a direct connection to mathematics accessibility.

Accessibility has been a purposeful consideration from the very beginning of the MathML specification, as alluded to in the 1998 MathML 1.0 specification. This understanding is further reflected in the very first version of the Web Content Accessibility Guidelines (WCAG 1.0, W3C Recommendation 5-May-1999), which mentions the use of MathML as a suggested technique to comply with Checkpoint 3.1, "When an appropriate markup language exists, use markup rather than images to convey information," by including the example technique to "use MathML to mark up mathematical equations..." It is also worth noting, that under the discussion of WCAG 1.0 Guideline 3, "Use markup and style sheets and do so properly," that the editors have included the admonition that "content developers must not sacrifice appropriate markup because a certain browser or assistive technology does not process it correctly." Now some 20 years after the publication of the original WCAG recommendation, we still struggle with the fact that many content developers have been slow to adopt MathML due to those very reasons. However, with the publication of MathML 4.0, the accessibility community is hopeful of what the future will bring for widespread mathematics accessibility on the web.

Using MathML in digital content extends the potential to support a wide array of accessibility use cases. We discuss these below.

Auditory output. Technological means of providing dynamic text-to-speech output for mathematical expressions precedes the origins of MathML, and this use case has had an impact on shaping the MathML specification from the beginning. Beyond simply generating spoken text strings, the use of audio cues such as changes in spoken pitch to help provide an auditory analog of two-dimensional visual structure has been found useful. Other audio applications have included other types of audio cues such as binaural spatialization, earcons, and spearcons to help disambiguate mathematical expressions rendered by synthetic speech. MathML provides a level of robust information about the structure and syntax of mathematical expressions to enable these techniques. It is also important to note that the ability to create extensive sets of automated speech rules used by MathML-aware TTS tools provide for virtually infinite portability of math speech to the various human spoken languages (e.g., internationalization), as well as different styles of spoken math (e.g., ClearSpeak, MathSpeak, SimpleSpeak, etc.). In the future, this could provide even more types of speech rules, such as when an educational assessment needs to apply a more restrictive reading so as not to invalidate the testing construct, or when instructional content aimed at early learners needs to adopt the spoken style used in the classroom for young students.

Braille output. The tactile rendering of mathematical expressions in braille is a very important use case. For someone who is blind, interpreting mathematics through auditory rendering alone is a cognitive taxing experience except for the most basic expressions. And for a deafblind user, auditory renderings are completely inaccessible. Several math braille codes are in common use globally, such as the Nemeth braille code, UEB Technical, German braille mathematics code, French braille mathematics code, etc. Dynamic mathematics braille translators such as Liblouis support translation of MathML content on webpages for individuals who access the web via a refreshable braille display. Thus, using MathML is essential for providing dynamic braille content for mathematics.

Other forms of visual transformation. Synchronized highlighting is a common addition to text-to-speech intended for sighted users. Because MathML provides the ability to parse the underlying tree structure of expressions, individual elements of the expression can be visually highlighted as they are spoken. This enhances the ability of TTS users to stay engaged with the text reading, which can potentially increase comprehension and learning. Even for people visually reading without TTS, visual highlighting within expressions as one navigates a web page using caret browsing can be a useful accessibility feature which MathML can potentially support.

For individuals who are deaf or hard of hearing but are unable to use braille, mathematical equations rendered in MathML can potentially be turned into visually displayed text. Since research has shown that, especially among school-age children with reading impairments, the ability to understand symbolic notation occurring in mathematical expression is much more difficult than reading literary text, enabling this capability could be a useful access technique for this population.

Another potential accessibility scaffold which MathML could provide for individuals who are deaf or hard of hearing would be the ability to provide input to automated signing avatars. Automated signing avatar technology which generates American Sign Language has already been applied to elementary level mathematics add citation. Sign languages vary by county (and sometimes locality) and are not simply "word to sign" translations, as sign language has its own grammar, so being able to access the underlying tree structure of mathematical expressions as can be done with MathML will provide the potential for representing expressions in sign language from a digital document dynamically without having to use static prerecorded videos of human signers.

Graphing an equation is a commonly used means of generating a visual output which can aid in comprehending the effects and implications of the underlying mathematical expressions. This is helpful for all people, but can be especially impactful for those with cognitive or learning impairments. Some dynamic graphing utilities (e.g., Desmos and MathTrax) have extended this concept beyond a simple visual line trace, to auditory tracing (e.g., tones which rise and fall in pitch to provide an audio construct of the visual trace) as well as a dynamically generated text description of the visual graph. Using MathML in digital content will provide the potential for developers to apply such automated accessible graphing utilities to their websites.

C.3 Accessibility Guidance

C.3.1 User Agents

C.3.1.1 Accessibility tree

User agents (e.g., web browsers) should leverage information in the MathML expression tree structure to maximize accessibility. Browsers should process MathML into the DOM tree's internal representation, which contains objects representing all the markup's elements and attributes. In general, user agents will expose accessibility information via a platform accessibility service (e.g., an accessibility API), which is passed on to assistive technology applications via the accessibility tree. The accessibility tree should contain accessibility-related information for most MathML elements. Browsers should ensure the accessibility tree generated from the DOM tree retains this information so that Accessibility APIs can provide a representation that can be understood by assistive technologies. However, in compliance with the W3C User Agent Accessibility Guidelines Success Criterion 4.1.4, "If the user agent accessibility API does not provide sufficient information to one or more platform accessibility services, then Document Object Models (DOM), must be made programmatically available to assistive technologies" [UAAG20].

By ensuring that most MathML elements become nodes in the DOM tree, and the resulting accessibility tree, user agents can expose math nodes for keyboard navigation within expressions. This can support important user needs such as the ability to visually highlight elements of an expression and/or speak individual elements as one navigates with arrow keys. This can further support other forms of synchronous navigation, such as individuals using refreshable braille displays along with synthetic speech.

While it is common practice for the accessibility tree to ignore most DOM node elements that are primarily used for visual display purposes, it is important to point out that math expressions often use what appears as visual styling to convey information which can be important for some types of assistive technology applications. For example, omitting the `<mspace>` element from the accessibility tree will impact the ability to generate a valid math braille representation of expressions on a braille display. Further, when color is expressed in MathML with the `mathcolor` and `mathbackground` attributes, these elements need to be included if they are used to express meaning.

The `alttext` attribute can be used to override standard speech rule processing (e.g., as is often done in standardized assessments). However, there are numerous limitations to this method. For instance, the entire spoken text of the expression must be given in the tag, even if the author is only concerned about one small portion. Further, `alttext` is limited to plain text, so speech queues such as pausing and pitch changes cannot be included for passing on to speech engines. Also, the `alttext` attribute has no direct linkage to the MathML tree, so there will be no way to handle synchronized highlighting of the expression, nor will there be a way for users to navigate through an expression.

An early draft of *MathML Accessiblity API Mappings 1.0* is available. This specification is intended for user agent developers responsible for MathML accessibility in their product. The goal of this specification is to maximize the accessibility of MathML content by ensuring each assistive technology receives MathML content with the roles, states, and properties it expects. The placing of ARIA labels and `aria-labeledby` is not appropriate in MathML because this will override braille generation.

C.4 Content Authors

This section considers how to use WCAG to establish requirements for accessible MathML content on the web, using the same four high-level content principles: that content should be perceivable, operable, understandable, and robust. Therefore, this section defines how to apply the conformance criteria defined in WCAG to address qualities unique to digital content containing MathML.

C.4.1 Overarching guidance

C.4.1.1 Always use markup

It is important that MathML be used for marking up all mathematics and linear chemical equation content. This precludes simply using ASCII characters or expression images in HTML (even if alt text is used). Even a single letter variable ideally should be marked up in MathML because it represents a mathematical expression. This way, audio, braille and visual renderings of the variable will be consistent throughout the page.

C.4.1.2 Use `intent` and `arg` attributes

MathML's `intent` and `arg` attributes has been developed to reduce notational ambiguity which cannot be reliably resolved by assistive technology. This also includes blanks and units, which are covered by the Intent attribute.

C.4.2 Specific Markup Guidance

C.4.2.1 Invisible Operators

Common use of mathematical notation employs several “invisible operators” whose symbols are not displayed but function as if the visible operator were present. These operators should be marked up in MathML to preserve their meaning as well as to prevent possible ambiguity for users of assistive technology.

Screen readers will not speak anything enclosed in an `<mphantom>` element; therefore, do not use `<mphantom>` in combination with an operator to create invisible operators.

Implicit Multiplication: The “invisible times” operator (⁢) should be used to indicate multiplication whenever the multiplication operator is used tacitly in traditional notation.

Function Application: The "apply function" operator (⁡) should be used to indicate function application.

Invisible Comma: The “invisible comma” or “invisible separator” operator (⁣) should be used to semantically

separate arguments or indices when commas are omitted.

Implicit Addition: In mixed fractions the “invisible plus” character (⁤) should be used as an operator between the whole number and its fraction.

C.4.2.2 Proper Grouping of Sub-expressions

It is good practice to group sub-expressions as they would be interpreted mathematically. Properly grouping sub-expressions using `<mrow>` can improve display by affecting spacing, allows for more intelligent linebreaking and indentation, it can simplify semantic interpretation of presentation elements by screen readers and text-to-speech applications.

C.4.2.3 Spacing

In general, the spacing elements `<mspace>`, `<mphantom>`, and `<mpadded>` should not be used to convey meaning.

C.4.2.4 Numbers

All numeric quantities should be enclosed in an `<mn>` element. Digit group separators, such as commas, periods, or spaces, should also be included as part of the number and should not be treated as operators.

C.4.2.5 Superscripts and Subscripts

It is important to apply superscripts and subscripts to the appropriate element or sub-expression. It is not correct to apply a superscript or subscript to a closing parenthesis or any other grouping symbol. Important for navigation

C.4.2.6 Elementary Math Notation

Elementary notations have their own layout elements. For long division and stacked expressions use the proper elements such as `<mlongdiv>` and `<mstack>` instead of `<mtable>`.

C.4.2.7 Fill-in-the-Blanks

Blanks in a “fill-in-the-blank” style of question are often visualized by underlined spaces, empty circles, squares, or other symbols. To indicate a blank, use the `intent` and `arg` attributes.

In an interactive electronic environment where the user should fill the blank on the displayed page, JavaScript would typically be used to invoke an editor when the blank is clicked on. To facilitate this, an `id` should be added to the element to identify it for editing and eventual processing. Additionally, an `onclick` or similar event trigger should be added. The details depend upon the type of interaction desired, along with the specific JavaScript being used.

C.4.2.8 Tables and Lists

MathML provides built-in support for tables and equation numbering, which complements HTML functionality with lists and tables. In practice, it is not always clear which structural elements should be used. Ideally, a table (either HTML `<table>` or MathML `<mtable>`) should be used when information between aligned rows or columns are semantically related. In other cases, such as ordinary problem numbering or information presented in an ordered sequence, an HTML ordered list `<ol>`; is more appropriate.

Choosing between `<table>` and `<mtable>` may require some forethought in how best to meet the usability needs of the intended audience and purpose of the table content. HTML structural elements are advantageous because screen readers provide more robust table navigation, whereas the user may only "enter" or "exit" an `<mtable>` in a MathML island. However, the `<mtable>` element is useful because it can be tweaked easily for visual alignment without creating new table cells, which can improve reading flow for the user. However, `<mtable>` should still be used for matrices and other table-like math layouts.

C.4.2.9 Natural-language Mathematics

Instructional content for young learners may sometimes use the written form of math symbols. For example, the multiplication sign × might be written as “times” or “multiplied by”. Because “times” and “multiplied by” are ordinary words, speech engines will not have an issue reading them. However, in some cases, there may be a use-case for including these terms in MathML. For instance, the word “times” in “x = 2 times a” could be marked up as an operator by means of <mo>times</mo>.

D. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *SHOULD*, and *SHOULD NOT* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Information nowadays is commonly generated, processed and rendered by software tools. The exponential growth of the Web is fueling the development of advanced systems for automatically searching, categorizing, and interconnecting information. In addition, there are increasing numbers of Web services, some of which offer technically based materials and activities. Thus, although MathML can be written by hand and read by humans, whether machine-aided or just with much concentration, the future of MathML is largely tied to the ability to process it with software tools.

There are many different kinds of MathML processors: editors for authoring MathML expressions, translators for converting to and from other encodings, validators for checking MathML expressions, computation engines that evaluate, manipulate, or compare MathML expressions, and rendering engines that produce visual, aural, or tactile representations of mathematical notation. What it means to support MathML varies widely between applications. For example, the issues that arise with a validating parser are very different from those for an equation editor.

This section gives guidelines that describe different types of MathML support and make clear the extent of MathML support in a given application. Developers, users, and reviewers are encouraged to use these guidelines in characterizing products. The intention behind these guidelines is to facilitate reuse by and interoperability of MathML applications by accurately setting out their capabilities in quantifiable terms.

The W3C Math Working Group maintains MathML Compliance Guidelines. Consult this document for future updates on conformance activities and resources.

D.1 MathML Conformance

A valid MathML expression is an XML construct determined by the MathML RelaxNG Schema together with the additional requirements given in this specification.

We shall use the phrase “a MathML processor” to mean any application that can accept or produce a valid MathML expression. A MathML processor that both accepts and produces valid MathML expressions may be able to “round-trip” MathML. Perhaps the simplest example of an application that might round-trip a MathML expression would be an editor that writes it to a new file without modifications.

Three forms of MathML conformance are specified:

1. A MathML-input-conformant processor must accept all valid MathML expressions; it should appropriately translate all MathML expressions into application-specific form allowing native application operations to be performed.

2. A MathML-output-conformant processor must generate valid MathML, appropriately representing all application-specific data.

3. A MathML-round-trip-conformant processor must preserve MathML equivalence. Two MathML expressions are “equivalent” if and only if both expressions have the same interpretation (as stated by the MathML Schema and specification) under any relevant circumstances, by any MathML processor. Equivalence on an element-by-element basis is discussed elsewhere in this document.

Beyond the above definitions, the MathML specification makes no demands of individual processors. In order to guide developers, the MathML specification includes advisory material; for example, there are many recommended rendering rules throughout 3. Presentation Markup. However, in general, developers are given wide latitude to interpret what kind of MathML implementation is meaningful for their own particular application.

To clarify the difference between conformance and interpretation of what is meaningful, consider some examples:

1. In order to be MathML-input-conformant, a validating parser needs only to accept expressions, and return “true” for expressions that are valid MathML. In particular, it need not render or interpret the MathML expressions at all.

2. A MathML computer-algebra interface based on content markup might choose to ignore all presentation markup. Provided the interface accepts all valid MathML expressions including those containing presentation markup, it would

be technically correct to characterize the application as MathML-input-conformant.

3. An equation editor might have an internal data representation that makes it easy to export some equations as MathML but not others. If the editor exports the simple equations as valid MathML, and merely displays an error message to the effect that conversion failed for the others, it is still technically MathML-output-conformant.

D.1.1 MathML Test Suite and Validator

As the previous examples show, to be useful, the concept of MathML conformance frequently involves a judgment about what parts of the language are meaningfully implemented, as opposed to parts that are merely processed in a technically correct way with respect to the definitions of conformance. This requires some mechanism for giving a quantitative statement about which parts of MathML are meaningfully implemented by a given application. To this end, the W3C Math Working Group has provided a test suite.

The test suite consists of a large number of MathML expressions categorized by markup category and dominant MathML element being tested. The existence of this test suite makes it possible, for example, to characterize quantitatively the hypothetical computer algebra interface mentioned above by saying that it is a MathML-input-conformant processor which meaningfully implements MathML content markup, including all of the expressions in the content markup section of the test suite.

Developers who choose not to implement parts of the MathML specification in a meaningful way are encouraged to itemize the parts they leave out by referring to specific categories in the test suite.

For MathML-output-conformant processors, information about currently available tools to validate MathML is maintained at the W3C MathML Validator. Developers of MathML-output-conformant processors are encouraged to verify their output using this validator.

Customers of MathML applications who wish to verify claims as to which parts of the MathML specification are implemented by an application are encouraged to use the test suites as a part of their decision processes.

D.1.2 Deprecated MathML 1.x and MathML 2.x Features

MathML 4.0 contains a number of features of earlier MathML which are now deprecated. The following points define what it means for a feature to be deprecated, and clarify the relation between deprecated features and current MathML conformance.

1. In order to be MathML-output-conformant, authoring tools may not generate MathML markup containing deprecated features.

2. In order to be MathML-input-conformant, rendering and reading tools must support deprecated features if they are to be in conformance with MathML 1.x or MathML 2.x. They do not have to support deprecated features to be considered in conformance with MathML 4.0. However, all tools are encouraged to support the old forms as much as possible.

3. In order to be MathML-round-trip-conformant, a processor need only preserve MathML equivalence on expressions containing no deprecated features.

D.1.3 MathML Extension Mechanisms and Conformance

MathML 4.0 defines three basic extension mechanisms: the `mglyph` element provides a way of displaying glyphs for non-Unicode characters, and glyph variants for existing Unicode characters; the `maction` element uses attributes from other namespaces to obtain implementation-specific parameters; and content markup makes use of the `definitionURL` attribute, as well as Content Dictionaries and the `cd` attribute, to point to external definitions of mathematical semantics.

These extension mechanisms are important because they provide a way of encoding concepts that are beyond the scope of MathML 4.0 as presently explicitly specified, which allows MathML to be used for exploring new ideas not yet susceptible to standardization. However, as new ideas take hold, they may become part of future standards. For example, an emerging character that must be represented by an `mglyph` element today may be assigned a Unicode code point in the future. At that time, representing the character directly by its Unicode code point would be preferable. This transition into Unicode has already taken place for hundreds of characters used for mathematics.

Because the possibility of future obsolescence is inherent in the use of extension mechanisms to facilitate the discussion of new ideas, MathML can reasonably make no conformance requirements concerning the use of extension mechanisms, even when alternative standard markup is available. For example, using an `mglyph` element to represent an 'x' is permitted. However, authors and implementers are strongly encouraged to use standard markup whenever possible. Similarly, maintainers of documents employing MathML 4.0 extension mechanisms are encouraged to monitor relevant standards activity (e.g., Unicode, OpenMath, etc.) and to update documents as more standardized markup becomes available.

D.2 Handling of Errors

If a MathML-input-conformant application receives input containing one or more elements with an illegal number or type of attributes or child schemata, it should nonetheless attempt to render all the input in an intelligible way, i.e., to render normally those parts of the input that were valid, and to render error messages (rendered as if enclosed in an `merror` element) in place of invalid expressions.

MathML-output-conformant applications such as editors and translators may choose to generate `merror` expressions to signal errors in their input. This is usually preferable to generating valid, but possibly erroneous, MathML.

D.3 Attributes for unspecified data

The MathML attributes described in the MathML specification are intended to allow for good presentation and content markup. However it is never possible to cover all users' needs for markup. Ideally, the MathML attributes should be an open-ended list so that users can add specific attributes for specific renderers. However, this cannot be done within the confines of a single XML DTD or in a Schema. Although it can be done using extensions of the standard DTD, say, some authors will wish to use non-standard attributes to take advantage of renderer-specific capabilities while remaining strictly in conformance with the standard DTD.

To allow this, the MathML 1.0 specification *Mathematical Markup Language (MathML) 1.0 Specification* allowed the attribute `other` on all elements, for use as a hook to pass on renderer-specific information. In particular, it was intended as a hook for passing information to audio renderers, computer algebra systems, and for pattern matching in future macro/

extension mechanisms. The motivation for this approach to the problem was historical, looking to PostScript, for example, where comments are widely used to pass information that is not part of PostScript.

In the next period of evolution of MathML the development of a general XML namespace mechanism seemed to make the use of the `other` attribute obsolete. In MathML 2.0, the `other` attribute is deprecated in favor of the use of namespace prefixes to identify non-MathML attributes. The `other` attribute has been removed in MathML 4.0. although it is still valid (with no defined behavior) in the mathml4-legacy schema.

For example, in MathML 1.0, it was recommended that if additional information was used in a renderer-specific implementation for the `maction` element (3.7.1 Bind Action to Sub-Expression), that information should be passed in using the `other` attribute:

```
<maction actiontype="highlight" other="color='#ff0000'"> expression </maction>
```

From MathML 4.0 onwards, a `data-*` attribute could be used:

```
<body>
  ...
  <maction actiontype="highlight" data-color="#ff0000"> expression </maction>
  ...
</body>
```

Note that the intent of allowing non-standard attributes is *not* to encourage software developers to use this as a loophole for circumventing the core conventions for MathML markup. Authors and applications should use non-standard attributes judiciously.

D.4 Privacy Considerations

Web platform implementations of MathML should implement [MathML-Core], and so the Privacy Considerations specified there apply.

D.5 Security Considerations

Web platform implementations of MathML should implement [MathML-Core], and so the Security Considerations specified there apply.

In some situations, MathML expressions can be parsed as XML. The security considerations of XML parsing apply then as explained in [RFC7303].

E. The Content MathML Operators

The following tables summarize key syntax information about the Content MathML operator elements.

E.1 The Content MathML Constructors

The following table gives the child element syntax for container elements that correspond to constructor symbols. See 4.3.1 Container Markup for details and examples.

The **Name** of the element is in the first column, and provides a link to the section that describes the constructor.

The **Content** column gives the child elements that may be contained within the constructor.

Name	Content
set	`ContExp*`
list	`ContExp*`
vector	`ContExp*`
matrix	`ContExp*`
matrixrow	`ContExp*`
lambda	`ContExp`
interval	`ContExp,ContExp`
piecewise	`piece*, otherwise?`
piece	`ContExp,ContExp`
otherwise	`ContExp`

E.2 The Content MathML Attributes

The following table lists the attributes that may be supplied on specific operator elements. In addition, all operator elements allow the `CommonAtt` and `DefEncAtt` attributes.

The **Name** of the element is in the first column, and provides a link to the section that describes the operator.

The **Attribute** column specifies the name of the attribute that may be supplied on the operator element.

The **Values** column specifies the values that may be supplied for the attribute specific to the operator element.

Name	Attribute	Values
tendsto	`type?`	string
interval	`closure?`	`open` | `closed` | `open-closed` | `closed-open`
set	`type?`	`set` | `multiset` | text
list	`order`	`numeric` | `lexicographic`

E.3 The Content MathML Operators

The **Name** of the element is in the first column, and provides a link to the section that describes the operator.

The **Symbol(s)** column provides a list of `csymbol`s that may be used to encode the operator, with links to the OpenMath

symbols used in the Strict Content MathML Transformation Algorithm.

The **Class** column specifies the operator class, which indicates how many arguments the operator expects, and may determine the mapping to Strict Content MathML, as described in 4.3.4 Operator Classes.

The **Qualifiers** column lists the qualifier elements accepted by the operator, either as child elements (for container elements) or as following sibling elements (for empty operator elements).

Name	Symbol(s)	Class	Qualifiers
plus	plus	nary-arith	BvarQ,DomainQ
times	times	nary-arith	BvarQ,DomainQ
gcd	gcd	nary-arith	BvarQ,DomainQ
lcm	lcm	nary-arith	BvarQ,DomainQ
compose	left_compose	nary-functional	BvarQ,DomainQ
and	and	nary-logical	BvarQ,DomainQ
or	or	nary-logical	BvarQ,DomainQ
xor	xor	nary-logical	BvarQ,DomainQ
selector	vector_selector, matrix_selector	nary-linalg	
union	union	nary-set	BvarQ,DomainQ
intersect	intersect	nary-set	BvarQ,DomainQ
cartesianproduct	cartesian_product	nary-set	BvarQ,DomainQ
vector	vector	nary-constructor	BvarQ,DomainQ
matrix	matrix	nary-constructor	BvarQ,DomainQ
matrixrow	matrixrow	nary-constructor	BvarQ,DomainQ
eq	eq	nary-reln	BvarQ,DomainQ
gt	gt	nary-reln	BvarQ,DomainQ
lt	lt	nary-reln	BvarQ,DomainQ
geq	geq	nary-reln	BvarQ,DomainQ
leq	leq	nary-reln	BvarQ,DomainQ
subset	subset	nary-set-reln	
prsubset	prsubset	nary-set-reln	
max	max	nary-minmax	BvarQ,DomainQ
min	min	nary-minmax	BvarQ,DomainQ
mean	mean, mean	nary-stats	BvarQ,DomainQ
median	median	nary-stats	BvarQ,DomainQ
mode	mode	nary-stats	BvarQ,DomainQ
sdev	sdev, sdev	nary-stats	BvarQ,DomainQ
variance	variance, variance	nary-stats	BvarQ,DomainQ
quotient	quotient	binary-arith	
divide	divide	binary-arith	
minus minus	unary_minus, minus	unary-arith, binary-arith	
power	power	binary-arith	
rem	remainder	binary-arith	
root root	root	unary-arith, binary-arith	degree
implies	implies	binary-logical	
equivalent	equivalent	binary-logical	BvarQ,DomainQ
neq	neq	binary-reln	
approx	approx	binary-reln	
factorof	factorof	binary-reln	
tendsto	limit	binary-reln	
vectorproduct	vectorproduct	binary-linalg	
scalarproduct	scalarproduct	binary-linalg	
outerproduct	outerproduct	binary-linalg	
in	in	binary-set	
notin	notin	binary-set	
notsubset	notsubset	binary-set	
notprsubset	notprsubset	binary-set	
setdiff	setdiff, setdiff	binary-set	
not	not	unary-logical	
factorial	factorial	unary-arith	
minus minus	unary_minus, minus	unary-arith, binary-arith	
root root	root	unary-arith, binary-arith	degree
abs	abs	unary-arith	
conjugate	conjugate	unary-arith	

F. The Strict Content MathML Transformation

MathML assigns semantics to content markup by defining a mapping to Strict Content MathML. Strict MathML, in turn, is in one-to-one correspondence with OpenMath, and the subset of OpenMath expressions obtained from content MathML expressions in this fashion all have well-defined semantics via the standard OpenMath Content Dictionary set. Consequently, the mapping of arbitrary content MathML expressions to equivalent Strict Content MathML plays a key role in underpinning the meaning of content MathML.

The mapping of arbitrary content MathML into Strict content MathML is defined algorithmically. The algorithm is described below as a collection of rewrite rules applying to specific non-Strict constructions. The individual rewrite transformations are described in the following subsections. The goal of this section is to outline the complete algorithm in one place.

The algorithm is a sequence of nine steps. Each step is applied repeatedly to rewrite the input until no further application is possible. Note that in many programming languages, such as XSLT, the natural implementation is as a recursive algorithm, rather than the multi-pass implementation suggested by the description below. The translation to XSL is straightforward and produces the same eventual Strict Content MathML. However, because the overall structure of the multi-pass algorithm is clearer, that is the formulation given here.

To transform an arbitrary content MathML expression into Strict Content MathML, apply each of the following rules in turn to the input expression until all instances of the target constructs have been eliminated:

1. *Rewrite non-strict* `bind` *and eliminate deprecated elements*: Change the outer `bind` tags in binding expressions to `apply` if they have qualifiers or multiple children. This simplifies the algorithm by allowing the subsequent rules to be applied to non-strict binding expressions without case distinction. Note that the later rules will change the `apply` elements introduced in this step back to `bind` elements.

2. *Apply special case rules for idiomatic uses of qualifiers*:

1. Rewrite derivatives with rules Rewrite: diff, Rewrite: nthdiff, and Rewrite: partialdiffdegree to explicate the binding status of the variables involved.
2. Rewrite integrals with the rules Rewrite: int, Rewrite: defint and Rewrite: defint limits to disambiguate the status of bound and free variables and of the orientation of the range of integration if it is given as a `lowlimit`/`uplimit` pair.
3. Rewrite limits as described in Rewrite: tendsto and Rewrite: limits condition.
4. Rewrite sums and products as described in 4.3.5.2 N-ary Sum `<sum/>` and 4.3.5.3 N-ary Product `<product/>`.
5. Rewrite roots as described in F.2.5 Roots.
6. Rewrite logarithms as described in F.2.6 Logarithms.
7. Rewrite moments as described in F.2.7 Moments.

3. *Rewrite Qualifiers to `domainofapplication`*: These rules rewrite all `apply` constructions using `bvar` and qualifiers to those using only the general `domainofapplication` qualifier.
   1. *Intervals*: Rewrite qualifiers given as `interval` and `lowlimit`/`uplimit` to intervals of integers via Rewrite: interval qualifier.
   2. *Multiple `conditions`*: Rewrite multiple `condition` qualifiers to a single one by taking their conjunction. The resulting compound `condition` is then rewritten to `domainofapplication` according to rule Rewrite: condition.
   3. *Multiple `domainofapplications`*: Rewrite multiple `domainofapplication` qualifiers to a single one by taking the intersection of the specified domains.
4. *Normalize Container Markup*:
   1. Rewrite sets and lists by the rule Rewrite: n-ary setlist domainofapplication.
   2. Rewrite interval, vectors, matrices, and matrix rows as described in F.3.1 Intervals, 4.3.5.8 N-ary Matrix Constructors: `<vector/>`, `<matrix/>`, `<matrixrow/>`. Note any qualifiers will have been rewritten to `domainofapplication` and will be further rewritten in Step 6.
   3. Rewrite lambda expressions by the rules Rewrite: lambda and Rewrite: lambda domainofapplication.
   4. Rewrite piecewise functions as described in 4.3.10.5 Piecewise declaration `<piecewise>`, `<piece>`, `<otherwise>`.
5. *Apply Special Case Rules for Operators using `domainofapplication` Qualifiers*: This step deals with the special cases for the operators introduced in 4.3 Content MathML for Specific Structures. There are different classes of special cases to be taken into account:
   1. Rewrite `min`, `max`, `mean` and similar n-ary/unary operators by the rules Rewrite: n-ary unary set, Rewrite: n-ary unary domainofapplication and Rewrite: n-ary unary single.
   2. Rewrite the quantifiers `forall` and `exists` used with `domainofapplication` to expressions using implication and conjunction by the rule Rewrite: quantifier.
   3. Rewrite integrals used with a `domainofapplication` element (with or without a `bvar`) according to the rules

Rewrite: int and Rewrite: defint.

4. Rewrite sums and products used with a `domainofapplication` element (with or without a `bvar`) as described in 4.3.5.2 N-ary Sum `<sum/>` and 4.3.5.3 N-ary Product `<product/>`.

6. *Eliminate `domainofapplication`*: At this stage, any `apply` has at most one `domainofapplication` child and special cases have been addressed. As `domainofapplication` is not Strict Content MathML, it is rewritten

   1. into an application of a restricted function via the rule Rewrite: restriction if the `apply` does not contain a `bvar` child.

   2. into an application of the predicate_on_list symbol via the rules Rewrite: n-ary relations and Rewrite: n-ary relations bvar if used with a relation.

   3. into a construction with the apply_to_list symbol via the general rule Rewrite: n-ary domainofapplication for general n-ary operators.

   4. into a construction using the suchthat symbol from the set1 content dictionary in an `apply` with bound variables via the Rewrite: apply bvar domainofapplication rule.

7. *Rewrite non-strict token elements*:

   1. Rewrite numbers represented as `cn` elements where the `type` attribute is one of `e-notation`, `rational`, `complex-cartesian`, `complex-polar`, `constant` as strict `cn` via rules Rewrite: cn sep, Rewrite: cn based_integer and Rewrite: cn constant.

   2. Rewrite any `ci`, `csymbol` or `cn` containing presentation MathML to `semantics` elements with rules Rewrite: cn presentation mathml and Rewrite: ci presentation mathml and the analogous rule for `csymbol`.

8. *Rewrite operators*: Rewrite any remaining operator defined in 4.3 Content MathML for Specific Structures to a `csymbol` referencing the symbol identified in the syntax table by the rule Rewrite: element. As noted in the descriptions of each operator element, some require special case rules to determine the proper choice of symbol. Some cases of particular note are:

   1. The order of the arguments for the `selector` operator must be rewritten, and the symbol depends on the type of the arguments.

   2. The choice of symbol for the `minus` operator depends on the number of the arguments, `minus` or `minus`.

   3. The choice of symbol for some set operators depends on the values of the `type` of the arguments.

   4. The choice of symbol for some statistical operators depends on the values of the types of the arguments.

9. *Rewrite non-strict attributes*:

   1. *Rewrite the `type` attribute*: At this point, all elements that accept the `type`, other than `ci` and `csymbol`, should have been rewritten into Strict Content Markup equivalents without `type` attributes, where type information is reflected in the choice of operator symbol. Now rewrite remaining `ci` and `csymbol` elements with a `type` attribute to a strict expression with `semantics` according to rules Rewrite: ci type annotation and Rewrite: csymbol type annotation.

   2. *Rewrite `definitionURL` and `encoding` attributes*: If the `definitionURL` and `encoding` attributes on a `csymbol` element can be interpreted as a reference to a content dictionary (see 4.2.3.2 Non-Strict uses of

<csymbol> for details), then rewrite to reference the content dictionary by the cd attribute instead.

3. *Rewrite attributes*: Rewrite any element with attributes that are not allowed in strict markup to a semantics construction with the element without these attributes as the first child and the attributes in annotation elements by rule Rewrite: attributes.

F.1 Rewrite non-strict bind

As described in 4.2.6 Bindings and Bound Variables <bind> and <bvar>, the strict form for the bind element does not allow qualifiers, and only allows one non-bvar child element.

Replace the bind tag in each binding expression with apply if it has qualifiers or multiple non-bvar child elements.

This step allows subsequent rules that modify non-strict binding expressions using apply to be used for non-strict binding expressions using bind without the need for a separate case.

Later rules will change these non-strict binding expressions using apply back to strict binding expressions using bind elements.

F.2 Rewrite idiomatic qualifiers

Apply special case rules for idiomatic uses of qualifiers.

F.2.1 Derivatives

Rewrite derivatives using the rules Rewrite: diff, Rewrite: nthdiff, and Rewrite: partialdiffdegree to make the binding status of the variables explicit.

For a differentiation operator it is crucial to realize that in the expression case, the variable is actually not bound by the differentiation operator.

Rewrite: diff

Translate an expression

```
<apply><diff/>
  <bvar><ci>x</ci></bvar>
  <ci>expression-in-x</ci>
</apply>
```

where <ci>expression-in-x</ci> is an expression in the variable *x* to the expression

```
<apply>
  <apply><csymbol cd="calculus1">diff</csymbol>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <ci>E</ci>
    </bind>
  </apply>
  <ci>x</ci>
</apply>
```

Note that the differentiated function is applied to the variable *x* making its status as a free variable explicit in strict markup. Thus the strict equivalent of

```
<apply><diff/>
  <bvar><ci>x</ci></bvar>
  <apply><sin/><ci>x</ci></apply>
</apply>
```

is

```
<apply>
  <apply><csymbol cd="calculus1">diff</csymbol>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <apply><csymbol cd="transc1">sin</csymbol><ci>x</ci></apply>
    </bind>
  </apply>
  <ci>x</ci>
</apply>
```

If the `bvar` element contains a `degree` element, use the `nthdiff` symbol.

Rewrite: nthdiff

```
<apply><diff/>
  <bvar><ci>x</ci><degree><ci>n</ci></degree></bvar>
  <ci>expression-in-x</ci>
</apply>
```

where `<ci>expression-in-x</ci>` is an expression in the variable *x* is translated to the expression

```
<apply>
  <apply><csymbol cd="calculus1">nthdiff</csymbol>
    <ci>n</ci>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <ci>expression-in-x</ci>
    </bind>
  </apply>
  <ci>x</ci>
</apply>
```

For example

```
<apply><diff/>
  <bvar><degree><cn>2</cn></degree><ci>x</ci></bvar>
  <apply><sin/><ci>x</ci></apply>
</apply>
```

Strict Content MathML equivalent

```
<apply>
  <apply><csymbol cd="calculus1">nthdiff</csymbol>
    <cn>2</cn>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <apply><csymbol cd="transc1">sin</csymbol><ci>x</ci></apply>
    </bind>
  </apply>
  <ci>x</ci>
</apply>
```

When applied to a function, the `partialdiff` element corresponds to the partialdiff symbol from the calculus1 content dictionary. No special rules are necessary as the two arguments of `partialdiff` translate directly to the two arguments of partialdiff.

Rewrite: partialdiffdegree

If `partialdiff` is used with an expression and `bvar` qualifiers it is rewritten to Strict Content MathML using the partialdiffdegree symbol.

```
<apply><partialdiff/>
  <bvar><ci>x1</ci><degree><ci>n1</ci></degree></bvar>
  <bvar><ci>xk</ci><degree><ci>nk</ci></degree></bvar>
  <degree><ci>total-n1-nk</ci></degree>
  <ci>expression-in-x1-xk</ci>
</apply>
```

where `<ci>expression-in-x1-xk</ci>` is an arbitrary expression involving the bound variables.

```
<apply>
  <apply><csymbol cd="calculus1">partialdiffdegree</csymbol>
    <apply><csymbol cd="list1">list</csymbol>
      <ci>n1</ci> <ci>nk</ci>
    </apply>
    <ci>total-n1-nk</ci>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x1</ci></bvar>
      <bvar><ci>xk</ci></bvar>
      <ci>expression-in-x1-xk</ci>
    </bind>
  </apply>
  <ci>x1</ci>
  <ci>xk</ci>
</apply>
```

If any of the bound variables do not use a `degree` qualifier, `<cn>1</cn>` should be used in place of the degree. If the original expression did not use the total degree qualifier then the second argument to partialdiffdegree should be the sum of the degrees. For example

```
<apply><csymbol cd="arith1">plus</csymbol>
  <ci>n1</ci> <ci>nk</ci>
</apply>
```

With this rule, the expression

```
<apply><partialdiff/>
  <bvar><ci>x</ci><degree><ci>n</ci></degree></bvar>
  <bvar><ci>y</ci><degree><ci>m</ci></degree></bvar>
  <apply><sin/>
    <apply><times/><ci>x</ci><ci>y</ci></apply>
  </apply>
</apply>
```

is translated into

```
<apply>
  <apply><csymbol cd="calculus1">partialdiffdegree</csymbol>
    <apply><csymbol cd="list1">list</csymbol>
      <ci>n</ci><ci>m</ci>
    </apply>
    <apply><csymbol cd="arith1">plus</csymbol>
      <ci>n</ci><ci>m</ci>
    </apply>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <bvar><ci>y</ci></bvar>
      <apply><csymbol cd="transc1">sin</csymbol>
        <apply><csymbol cd="arith1">times</csymbol>
          <ci>x</ci><ci>y</ci>
        </apply>
      </apply>
    </bind>
    <ci>x</ci>
    <ci>y</ci>
  </apply>
</apply>
```

F.2.2 Integrals

Rewrite integrals using the rules Rewrite: int, Rewrite: defint and Rewrite: defint limits to disambiguate the status of bound and free variables and of the orientation of the range of integration if it is given as a `lowlimit`/`uplimit` pair.

As an indefinite integral applied to a function, the `int` element corresponds to the int symbol from the calculus1 content dictionary. As a definite integral applied to a function, the `int` element corresponds to the defint symbol from the calculus1 content dictionary.

When no bound variables are present, the translation of an indefinite integral to Strict Content Markup is straight forward. When bound variables are present, the following rule should be used.

Rewrite: int

Translate an indefinite integral, where `<ci>expression-in-x</ci>` is an arbitrary expression involving the bound variable(s) `<ci>x</ci>`

```
<apply><int/>
  <bvar><ci>x</ci></bvar>
  <ci>expression-in-x</ci>
</apply>
```

to the expression

```
<apply>
  <apply><csymbol cd="calculus1">int</csymbol>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <ci>expression-in-x</ci>
    </bind>
  </apply>
  <ci>x</ci>
</apply>
```

Note that as *x* is not bound in the original indefinite integral, the integrated function is applied to the variable *x* making it an explicit free variable in Strict Content Markup expression, even though it is bound in the subterm used as an argument to int.

For instance, the expression

```
<apply><int/>
  <bvar><ci>x</ci></bvar>
  <apply><cos/><ci>x</ci></apply>
</apply>
```

has the Strict Content MathML equivalent

```
<apply>
  <apply><csymbol cd="calculus1">int</csymbol>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <apply><cos/><ci>x</ci></apply>
    </bind>
  </apply>
  <ci>x</ci>
</apply>
```

For a definite integral without bound variables, the translation is also straightforward.

For instance, the integral of a differential form *f* over an arbitrary domain *C* represented as

```
<apply><int/>
  <domainofapplication><ci>C</ci></domainofapplication>
  <ci>f</ci>
</apply>
```

is equivalent to the Strict Content MathML:

```
<apply><csymbol cd="calculus1">defint</csymbol><ci>C</ci><ci>f</ci></apply>
```

Note, however, the additional remarks on the translations of other kinds of qualifiers that may be used to specify a domain of integration in the rules for definite integrals following.

When bound variables are present, the situation is more complicated in general, and the following rules are used.

Rewrite: defint

Translate a definite integral, where `<ci>expression-in-x</ci>` is an arbitrary expression involving the bound variable(s) `<ci>x</ci>`

```
<apply><int/>
  <bvar><ci>x</ci></bvar>
  <domainofapplication><ci>D</ci></domainofapplication>
  <ci>expression-in-x</ci>
</apply>
```

to the expression

```
<apply><csymbol cd="calculus1">defint</csymbol>
  <ci>D</ci>
  <bind><csymbol cd="fns1">lambda</csymbol>
    <bvar><ci>x</ci></bvar>
    <ci>expression-in-x</ci>
  </bind>
</apply>
```

But the definite integral with a `lowlimit`/`uplimit` pair carries the strong intuition that the range of integration is oriented, and thus swapping lower and upper limits will change the sign of the result. To accommodate this, use the following special translation rule:

Rewrite: defint limits

```
<apply><int/>
  <bvar><ci>x</ci></bvar>
  <lowlimit><ci>a</ci></lowlimit>
  <uplimit><ci>b</ci></uplimit>
  <ci>expression-in-x</ci>
</apply>
```

where `<ci>expression-in-x</ci>` is an expression in the variable x is translated to the expression:

```
<apply><csymbol cd="calculus1">defint</csymbol>
  <apply><csymbol cd="interval1">oriented_interval</csymbol>
    <ci>a</ci> <ci>b</ci>
  </apply>
  <bind><csymbol cd="fns1">lambda</csymbol>
    <bvar><ci>x</ci></bvar>
    <ci>expression-in-x</ci>
  </bind>
</apply>
```

The oriented_interval symbol is also used when translating the `interval` qualifier, when it is used to specify the domain of integration. Integration is assumed to proceed from the left endpoint to the right endpoint.

The case for multiple integrands is treated analogously.

Note that use of the `condition` qualifier also requires special treatment. In particular, it extends to multivariate domains by using extra bound variables and a domain corresponding to a cartesian product as in:

```
<bind><int/>
  <bvar><ci>x</ci></bvar>
  <bvar><ci>y</ci></bvar>
  <condition>
    <apply><and/>
      <apply><leq/><cn>0</cn><ci>x</ci></apply>
      <apply><leq/><ci>x</ci><cn>1</cn></apply>
      <apply><leq/><cn>0</cn><ci>y</ci></apply>
      <apply><leq/><ci>y</ci><cn>1</cn></apply>
    </apply>
  </condition>
  <apply><times/>
    <apply><power/><ci>x</ci><cn>2</cn></apply>
    <apply><power/><ci>y</ci><cn>3</cn></apply>
  </apply>
</bind>
```

Strict Content MathML equivalent

```
<apply><csymbol cd="calculus1">defint</csymbol>
  <apply><csymbol cd="set1">suchthat</csymbol>
    <apply><csymbol cd="set1">cartesianproduct</csymbol>
      <csymbol cd="setname1">R</csymbol>
      <csymbol cd="setname1">R</csymbol>
    </apply>
    <apply><csymbol cd="logic1">and</csymbol>
      <apply><csymbol cd="arith1">leq</csymbol><cn>0</cn><ci>x</ci></apply>
      <apply><csymbol cd="arith1">leq</csymbol><ci>x</ci><cn>1</cn></apply>
      <apply><csymbol cd="arith1">leq</csymbol><cn>0</cn><ci>y</ci></apply>
      <apply><csymbol cd="arith1">leq</csymbol><ci>y</ci><cn>1</cn></apply>
    </apply>
    <bind><csymbol cd="fns11">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <bvar><ci>y</ci></bvar>
      <apply><csymbol cd="arith1">times</csymbol>
        <apply><csymbol cd="arith1">power</csymbol><ci>x</ci><cn>2</cn></apply>
        <apply><csymbol cd="arith1">power</csymbol><ci>y</ci><cn>3</cn></apply>
      </apply>
    </bind>
  </apply>
</apply>
```

F.2.3 Limits

Rewrite limits using the rules Rewrite: tendsto and Rewrite: limits condition.

The usage of `tendsto` to qualify a limit is formally defined by writing the expression in Strict Content MathML via the rule Rewrite: limits condition. The meanings of other more idiomatic uses of `tendsto` are not formally defined by this specification. When rewriting these cases to Strict Content MathML, `tendsto` should be rewritten to an annotated identifier as shown below.

Rewrite: tendsto

```
<tendsto/>
```

Strict Content MathML equivalent

```
<semantics>
  <ci>tendsto</ci>
  <annotation-xml encoding="MathML-Content">
    <tendsto/>
  </annotation-xml>
</semantics>
```

Rewrite: limits condition

```
<apply><limit/>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><tendsto/><ci>x</ci><cn>0</cn></apply>
  </condition>
  <ci>expression-in-x</ci>
</apply>
```

Strict Content MathML equivalent

```
<apply><csymbol cd="limit1">limit</csymbol>
  <cn>0</cn>
  <csymbol cd="limit1">null</csymbol>
  <bind><csymbol cd="fns1">lambda</csymbol>
    <bvar><ci>x</ci></bvar>
    <ci>expression-in-x</ci>
  </bind>
</apply>
```

where `<ci>expression-in-x</ci>` is an arbitrary expression involving the bound variable(s), and the choice of symbol, null, depends on the `type` attribute of the `tendsto` element as described in 4.3.10.4 Limits `<limit/>`.

F.2.4 Sums and Products

Rewrite sums and products as described in 4.3.5.2 N-ary Sum `<sum/>` and 4.3.5.3 N-ary Product `<product/>`.

When no explicit bound variables are used, no special rules are required to rewrite sums as Strict Content beyond the generic rules for rewriting expressions using qualifiers. However, when bound variables are used, it is necessary to introduce a `lambda` construction to rewrite the expression in the bound variables as a function.

Content MathML

```
<apply><sum/>
  <bvar><ci>i</ci></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <uplimit><cn>100</cn></uplimit>
  <apply><power/><ci>x</ci><ci>i</ci></apply>
</apply>
```

Strict Content MathML equivalent

```
<apply><csymbol cd="arith1">sum</csymbol>
  <apply><csymbol cd="interval1">integer_interval</csymbol>
    <cn>0</cn>
    <cn>100</cn>
  </apply>
  <bind><csymbol cd="fns1">lambda</csymbol>
    <bvar><ci>i</ci></bvar>
    <apply><csymbol cd="arith1">power</csymbol><ci>x</ci><ci>i</ci></apply>
  </bind>
</apply>
```

When no explicit bound variables are used, no special rules are required to rewrite products as Strict Content beyond the generic rules for rewriting expressions using qualifiers. However, when bound variables are used, it is necessary to introduce a `lambda` construction to rewrite the expression in the bound variables as a function.

Content MathML

```
<apply><product/>
  <bvar><ci>i</ci></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <uplimit><cn>100</cn></uplimit>
  <apply><power/><ci>x</ci><ci>i</ci></apply>
</apply>
```

Strict Content MathML equivalent

```
<apply><csymbol cd="arith1">product</csymbol>
  <apply><csymbol cd="interval1">integer_interval</csymbol>
    <cn>0</cn>
    <cn>100</cn>
  </apply>
  <bind><csymbol cd="fns1">lambda</csymbol>
    <bvar><ci>i</ci></bvar>
    <apply><csymbol cd="arith1">power</csymbol><ci>x</ci><ci>i</ci></apply>
  </bind>
</apply>
```

F.2.5 Roots

Rewrite roots as described in F.2.5 Roots.

In Strict Content markup, the root symbol is always used with two arguments, with the second indicating the degree of the root being extracted.

Content MathML

```
<apply><root/><ci>x</ci></apply>
```

Strict Content MathML equivalent

```
<apply><csymbol cd="arith1">root</csymbol>
  <ci>x</ci>
  <cn type="integer">2</cn>
</apply>
```

Content MathML

```
<apply><root/>
  <degree><ci type="integer">n</ci></degree>
  <ci>a</ci>
</apply>
```

Strict Content MathML equivalent

```
<apply><csymbol cd="arith1">root</csymbol>
  <ci>a</ci>
  <cn type="integer">n</cn>
</apply>
```

F.2.6 Logarithms

Rewrite logarithms as described in 4.3.7.9 Logarithm <log/> , <logbase>.

When mapping `log` to Strict Content, one uses the log symbol denoting the function that returns the log of its second argument with respect to the base specified by the first argument. When `logbase` is present, it determines the base. Otherwise, the default base of 10 must be explicitly provided in Strict markup. See the following example.

```
<apply><plus/>
  <apply>
    <log/>
    <logbase><cn>2</cn></logbase>
    <ci>x</ci>
  </apply>
  <apply>
    <log/>
    <ci>y</ci>
  </apply>
</apply>
```

Strict Content MathML equivalent:

```
<apply>
  <csymbol cd="arith1">plus</csymbol>
  <apply>
    <csymbol cd="transc1">log</csymbol>
    <cn>2</cn>
    <ci>x</ci>
  </apply>
  <apply>
    <csymbol cd="transc1">log</csymbol>
    <cn>10</cn>
    <ci>y</ci>
  </apply>
</apply>
```

F.2.7 Moments

Rewrite moments as described in 4.3.7.8 Moment <moment/>, <momentabout>.

When rewriting to Strict Markup, the moment symbol from the s_data1 content dictionary is used when the `moment` element is applied to an explicit list of arguments. When it is applied to a distribution, then the moment symbol from the s_dist1 content dictionary should be used. Both operators take the degree as the first argument, the point as the second, followed by the data set or random variable respectively.

```
<apply><moment/>
  <degree><cn>3</cn></degree>
  <momentabout><ci>p</ci></momentabout>
  <ci>X</ci>
</apply>
```

Strict Content MathML equivalent

```
<apply><csymbol cd="s_dist1">moment</csymbol>
  <cn>3</cn>
  <ci>p</ci>
  <ci>X</ci>
</apply>
```

F.3 Rewrite to `domainofapplication`

Rewrite Qualifiers to `domainofapplication`. These rules rewrite all `apply` constructions using `bvar` and qualifiers to those using only the general `domainofapplication` qualifier.

F.3.1 Intervals

Rewrite qualifiers given as `interval` and `lowlimit`/`uplimit` to intervals of integers via Rewrite: interval qualifier.

Rewrite: interval qualifier

```
<apply><ci>H</ci>
  <bvar><ci>x</ci></bvar>
  <lowlimit><ci>a</ci></lowlimit>
  <uplimit><ci>b</ci></uplimit>
  <ci>C</ci>
</apply>
```

```
<apply><ci>H</ci>
  <bvar><ci>x</ci></bvar>
  <domainofapplication>
    <apply><csymbol cd="interval1">interval</csymbol>
      <ci>a</ci>
      <ci>b</ci>
    </apply>
  </domainofapplication>
  <ci>C</ci>
</apply>
```

The symbol used in this translation depends on the head of the application, denoted by `<ci>H</ci>` here. By default interval should be used, unless the semantics of the head term can be determined and indicate a more specific interval symbol. In particular, several predefined Content MathML elements should be used with more specific interval symbols. If the head is `int` then oriented_interval is used. When the head term is `sum` or `product`, integer_interval should be used.

The above technique for replacing `lowlimit` and `uplimit` qualifiers with a `domainofapplication` element is also used for replacing the `interval` qualifier. Note that `interval` is only interpreted as a qualifier if it immediately follows `bvar`. In other contexts `interval` is interpreted as a constructor, F.4.2 Intervals, vectors, matrices.

F.3.2 Multiple `conditions`

Rewrite multiple `condition` qualifiers to a single one by taking their conjunction. The resulting compound `condition` is then rewritten to `domainofapplication` according to rule Rewrite: condition.

The `condition` qualifier restricts a bound variable by specifying a Boolean-valued expression on a larger domain, specifying whether a given value is in the restricted domain. The `condition` element contains a single child that represents the truth condition. Compound conditions are formed by applying Boolean operators such as `and` in the condition.

Rewrite: condition

To rewrite an expression using the `condition` qualifier as one using `domainofapplication`,

```
<bvar><ci>x1</ci></bvar>
<bvar><ci>xn</ci></bvar>
<condition><ci>P</ci></condition>
```

is rewritten to

```
<bvar><ci>x1</ci></bvar>
<bvar><ci>xn</ci></bvar>
<domainofapplication>
  <apply><csymbol cd="set1">suchthat</csymbol>
    <ci>R</ci>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x1</ci></bvar>
      <bvar><ci>xn</ci></bvar>
      <ci>P</ci>
    </bind>
  </apply>
</domainofapplication>
```

If the `apply` has a `domainofapplication` (perhaps originally expressed as `interval` or an `uplimit`/`lowlimit` pair) then that is used for `<ci>R</ci>`. Otherwise `<ci>R</ci>` is a set determined by the `type` attribute of the bound variable as specified in 4.2.2.2 Non-Strict uses of `<ci>`, if that is present. If the type is unspecified, the translation introduces an unspecified domain via content identifier `<ci>R</ci>`.

F.3.3 Multiple `domainofapplication`s

Rewrite multiple `domainofapplication` qualifiers to a single one by taking the intersection of the specified domains.

F.4 Normalize container markup

F.4.1 Sets and Lists

Rewrite sets and lists by the rule Rewrite: n-ary setlist domainofapplication.

The use of `set` and `list` follows the same format as other n-ary constructors, however when rewriting to Strict Content MathML a variant of the usual rule is used, since the map symbol implicitly constructs the required set or list, and apply_to_list is not needed in this case.

The elements representing these n-ary operators are specified in the schema pattern `nary-setlist-constructor.class`.

If the argument list is given explicitly, the Rewrite: element rule applies.

When qualifiers are used to specify the list of arguments, the following rule is used.

Rewrite: n-ary setlist domainofapplication

An expression of the following form, where `<set/>` is either of the elements `set` or `list` and `<ci>expression-in-x</ci>` is an arbitrary expression involving the bound variable(s)

```
<set>
  <bvar><ci>x</ci></bvar>
  <domainofapplication><ci>D</ci></domainofapplication>
  <ci>expression-in-x</ci>
</set>
```

is rewritten to

```
<apply><csymbol cd="set1">map</csymbol>
  <bind><csymbol cd="fns1">lambda</csymbol>
    <bvar><ci>x</ci></bvar>
    <ci>expression-in-x</ci>
  </bind>
  <ci>D</ci>
</apply>
```

Note that when `<ci>D</ci>` is already a set or list of the appropriate type for the container element, and the lambda function created from `<ci>expression-in-x</ci>` is the identity, the entire container element should be rewritten directly as `<ci>D</ci>`.

In the case of `set`, the choice of Content Dictionary and symbol depends on the value of the `type` attribute of the arguments. By default the set symbol is used, but if one of the arguments has `type` attribute with value `multiset`, the multiset symbol is used. If there is a `type` attribute with value other than `set` or `multiset` the set symbol should be used, and the arguments should be annotated with their type by rewriting the `type` attribute using the rule Rewrite: attributes.

F.4.2 Intervals, vectors, matrices

Rewrite interval, vectors, matrices, and matrix rows as described in F.3.1 Intervals, 4.3.5.8 N-ary Matrix Constructors: `<vector/>`, `<matrix/>`, `<matrixrow/>`. Note any qualifiers will have been rewritten to `domainofapplication` and will be further rewritten in a later step.

In Strict markup, the `interval` element corresponds to one of four symbols from the interval1 content dictionary. If `closure` has the value `open` then `interval` corresponds to the interval_oo. With the value `closed` `interval` corresponds to the symbol interval_cc, with value `open-closed` to interval_oc, and with `closed-open` to interval_co.

F.4.3 Lambda expressions

Rewrite lambda expressions by the rules Rewrite: lambda and Rewrite: lambda domainofapplication.

Rewrite: lambda

If the `lambda` element does not contain qualifiers, the lambda expression is directly translated into a `bind` expression.

```
<lambda>
  <bvar><ci>x1</ci></bvar><bvar><ci>xn</ci></bvar>
  <ci>expression-in-x1-xn</ci>
</lambda>
```

rewrites to the Strict Content MathML

```
<bind><csymbol cd="fns1">lambda</csymbol>
  <bvar><ci>x1</ci></bvar><bvar><ci>xn</ci></bvar>
  <ci>expression-in-x1-xn</ci>
</bind>
```

Rewrite: lambda domainofapplication

If the `lambda` element does contain qualifiers, the qualifier may be rewritten to `domainofapplication` and then the lambda expression is translated to a function term constructed with lambda and restricted to the specified domain using restriction.

```
<lambda>
  <bvar><ci>x1</ci></bvar><bvar><ci>xn</ci></bvar>
  <domainofapplication><ci>D</ci></domainofapplication>
  <ci>expression-in-x1-xn</ci>
</lambda>
```

rewrites to the Strict Content MathML

```
<apply><csymbol cd="fns1">restriction</csymbol>
  <bind><csymbol cd="fns1">lambda</csymbol>
    <bvar><ci>x1</ci></bvar><bvar><ci>xn</ci></bvar>
    <ci>expression-in-x1-xn</ci>
  </bind>
  <ci>D</ci>
</apply>
```

F.4.4 Piecewise functions

Rewrite piecewise functions as described in 4.3.10.5 Piecewise declaration <piecewise>, <piece>, <otherwise>.

In Strict Content MathML, the container elements `piecewise`, `piece` and `otherwise` are mapped to applications of the constructor symbols of the same names in the piece1 CD. Apart from the fact that these three elements (respectively symbols) are used together, the mapping to Strict markup is straightforward:

Content MathML

```
<piecewise>
  <piece>
    <cn>0</cn>
    <apply><lt/><ci>x</ci><cn>0</cn></apply>
  </piece>
  <piece>
    <cn>1</cn>
    <apply><gt/><ci>x</ci><cn>1</cn></apply>
  </piece>
  <otherwise>
    <ci>x</ci>
  </otherwise>
</piecewise>
```

Strict Content MathML equivalent

```
<apply><csymbol cd="piece1">piecewise</csymbol>
  <apply><csymbol cd="piece1">piece</csymbol>
    <cn>0</cn>
    <apply><csymbol cd="relation1">lt</csymbol><ci>x</ci><cn>0</cn></apply>
  </apply>
  <apply><csymbol cd="piece1">piece</csymbol>
    <cn>1</cn>
    <apply><csymbol cd="relation1">gt</csymbol><ci>x</ci><cn>1</cn></apply>
  </apply>
  <apply><csymbol cd="piece1">otherwise</csymbol>
    <ci>x</ci>
  </apply>
</apply>
```

F.5 Rewrite `domainofapplication` qualifiers

Apply Special Case Rules for Operators using `domainofapplication` Qualifiers. This step deals with the special cases for the operators introduced in 4.3 Content MathML for Specific Structures. There are different classes of special cases to be taken into account.

F.5.1 N-ary/unary operators

Rewrite `min`, `max`, `mean` and similar n-ary/unary operators by the rules Rewrite: n-ary unary set, Rewrite: n-ary unary domainofapplication and Rewrite: n-ary unary single.

Rewrite: n-ary unary set

When an element, `<max/>`, of class nary-stats or nary-minmax is applied to an explicit list of 0 or 2 or more arguments,

```
<ci>a1</ci><ci>a2</ci><ci>an</ci>
```

```
<apply><max/><ci>a1</ci><ci>a2</ci><ci>an</ci></apply>
```

it is translated to the unary application of the symbol `<csymbol cd="minmax1" name="max"/>` as specified in the syntax table for the element to the set of arguments, constructed using the `<csymbol cd="set1" name="set"/>` symbol.

```
<apply><csymbol cd="minmax1">max</csymbol>
  <apply><csymbol cd="set1">set</csymbol>
    <ci>a1</ci><ci>a2</ci><ci>an</ci>
  </apply>
</apply>
```

Like all MathML n-ary operators, the list of arguments may be specified implicitly using qualifier elements. This is expressed in Strict Content MathML using the following rule, which is similar to the rule Rewrite: n-ary domainofapplication but differs in that the symbol can be directly applied to the constructed set of arguments and it is not necessary to use apply_to_list.

Rewrite: n-ary unary domainofapplication

An expression of the following form, where `<max/>` represents any element of the relevant class and `<ci>expression-in-x</ci>` is an arbitrary expression involving the bound variable(s)

```
<apply><max/>
  <bvar><ci>x</ci></bvar>
  <domainofapplication><ci>D</ci></domainofapplication>
  <ci>expression-in-x</ci>
</apply>
```

is rewritten to

```
<apply><csymbol cd="minmax1">max</csymbol>
  <apply><csymbol cd="set1">map</csymbol>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <ci>expression-in-x</ci>
    </bind>
    <ci>D</ci>
  </apply>
</apply>
```

Note that when `<ci>D</ci>` is already a set and the lambda function created from `<ci>expression-in-x</ci>` is the identity, the `domainofapplication` term should be rewritten directly as `<ci>D</ci>`.

If the element is applied to a single argument the set symbol is not used and the symbol is applied directly to the argument.

Rewrite: n-ary unary single

When an element, `<max/>`, of class nary-stats or nary-minmax is applied to a single argument,

```
<apply><max/><ci>a</ci></apply>
```

it is translated to the unary application of the symbol in the syntax table for the element.

```
<apply><csymbol cd="minmax1">max</csymbol> <ci>a</ci> </apply>
```

Note: Earlier versions of MathML were not explicit about the correct interpretation of elements in this class, and left it undefined as to whether an expression such as max(X) was a trivial application of max to a singleton, or whether it should be interpreted as meaning the maximum of values of the set X. Applications finding that the rule Rewrite: n-ary unary single can not be applied as the supplied argument is a scalar may wish to use the rule Rewrite: n-ary unary set as an error recovery. As a further complication, in the case of the statistical functions the Content Dictionary to use in this case depends on the desired interpretation of the argument as a set of explicit data or a random variable representing a distribution.

F.5.2 Quantifiers

Rewrite the quantifiers `forall` and `exists` used with `domainofapplication` to expressions using implication and conjunction by the rule Rewrite: quantifier.

If used with `bind` and no qualifiers, then the interpretation in Strict Content MathML is simple. In general if used with `apply` or qualifiers, the interpretation in Strict Content MathML is via the following rule.

Rewrite: quantifier

An expression of following form where `<exists/>` denotes an element of class `quantifier` and `<ci>expression-in-x</ci>` is an arbitrary expression involving the bound variable(s)

```
<apply><exists/>
  <bvar><ci>x</ci></bvar>
  <domainofapplication><ci>D</ci></domainofapplication>
  <ci>expression-in-x</ci>
</apply>
```

is rewritten to an expression

```
<bind><csymbol cd="quant1">exists</csymbol>
  <bvar><ci>x</ci></bvar>
  <apply><csymbol cd="logic1">and</csymbol>
    <apply><csymbol cd="set1">in</csymbol><ci>x</ci><ci>D</ci></apply>
    <ci>expression-in-x</ci>
  </apply>
</bind>
```

where the symbols `<csymbol cd="quant1">exists</csymbol>` and `<csymbol cd="logic1">and</csymbol>` are as specified in the syntax table of the element. (The additional symbol being and in the case of `exists` and implies in the case of `forall`.) When no `domainofapplication` is present, no logical conjunction is necessary, and the translation is direct.

When the `forall` element is used with a `condition` qualifier the strict equivalent is constructed with the help of logical implication by the rule Rewrite: quantifier. Thus

```
<bind><forall/>
  <bvar><ci>p</ci></bvar>
  <bvar><ci>q</ci></bvar>
  <condition>
    <apply><and/>
      <apply><in/><ci>p</ci><rationals/></apply>
      <apply><in/><ci>q</ci><rationals/></apply>
      <apply><lt/><ci>p</ci><ci>q</ci></apply>
    </apply>
  </condition>
  <apply><lt/>
    <ci>p</ci>
    <apply><power/><ci>q</ci><cn>2</cn></apply>
  </apply>
</bind>
```

translates to

```
<bind><csymbol cd="quant1">forall</csymbol>
  <bvar><ci>p</ci></bvar>
  <bvar><ci>q</ci></bvar>
  <apply><csymbol cd="logic1">implies</csymbol>
    <apply><csymbol cd="logic1">and</csymbol>
      <apply><csymbol cd="set1">in</csymbol>
        <ci>p</ci>
        <csymbol cd="setname1">Q</csymbol>
      </apply>
      <apply><csymbol cd="set1">in</csymbol>
        <ci>q</ci>
        <csymbol cd="setname1">Q</csymbol>
      </apply>
      <apply><csymbol cd="relation1">lt</csymbol><ci>p</ci><ci>q</ci></apply>
    </apply>
    <apply><csymbol cd="relation1">lt</csymbol>
      <ci>p</ci>
      <apply><csymbol cd="arith1">power</csymbol>
        <ci>q</ci>
        <cn>2</cn>
      </apply>
    </apply>
  </apply>
</bind>
```

F.5.3 Integrals

Rewrite integrals used with a `domainofapplication` element (with or without a `bvar`) according to the rules Rewrite: int and Rewrite: defint. See F.2.2 Integrals.

F.5.4 Sums and products

Rewrite sums and products used with a `domainofapplication` element (with or without a `bvar`) as described in 4.3.5.2 N-ary Sum `<sum/>` and 4.3.5.3 N-ary Product `<product/>`. See F.2.4 Sums and Products.

F.6 Eliminate `domainofapplication`

At this stage, any `apply` has at most one `domainofapplication` child and special cases have been addressed. As `domainofapplication` is not Strict Content MathML, it is rewritten as one of the following cases.

By applying the rules above, expressions using the `interval`, `condition`, `uplimit` and `lowlimit` can be rewritten using only `domainofapplication`. Once a `domainofapplication` has been obtained, the final mapping to Strict markup is

accomplished using the following rules:

F.6.1 Restricted function

Into an application of a restricted function via the rule Rewrite: restriction if the `apply` does not contain a `bvar` child.

Rewrite: restriction

An application of a function that is qualified by the `domainofapplication` qualifier (expressed by an `apply` element without bound variables) is converted to an application of a function term constructed with the restriction symbol.

```
<apply><ci>F</ci>
  <domainofapplication>
    <ci>C</ci>
  </domainofapplication>
  <ci>a1</ci>
  <ci>an</ci>
</apply>
```

may be written as:

```
<apply>
  <apply><csymbol cd="fns1">restriction</csymbol>
    <ci>F</ci>
    <ci>C</ci>
  </apply>
  <ci>a1</ci>
  <ci>an</ci>
</apply>
```

F.6.2 Predicate on list

Into an application of the predicate_on_list symbol via the rules Rewrite: n-ary relations and Rewrite: n-ary relations bvar if used with a relation.

Rewrite: n-ary relations

An expression of the form

```
<apply><lt/>
  <ci>a</ci><ci>b</ci><ci>c</ci><ci>d</ci>
</apply>
```

rewrites to Strict Content MathML

```
<apply><csymbol cd="fns2">predicate_on_list</csymbol>
  <csymbol cd="reln1">lt</csymbol>
  <apply><csymbol cd="list1">list</csymbol>
    <ci>a</ci><ci>b</ci><ci>c</ci><ci>d</ci>
  </apply>
</apply>
```

Rewrite: n-ary relations bvar

An expression of the form

```
<apply><lt/>
  <bvar><ci>x</ci></bvar>
  <domainofapplication><ci>R</ci></domainofapplication>
  <ci>expression-in-x</ci>
</apply>
```

where `<ci>expression-in-x</ci>` is an arbitrary expression involving the bound variable, rewrites to the Strict Content MathML

```
<apply><csymbol cd="fns2">predicate_on_list</csymbol>
  <csymbol cd="reln1">lt</csymbol>
  <apply><csymbol cd="list1">map</csymbol>
    <ci>R</ci>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <ci>expression-in-x</ci>
    </bind>
  </apply>
</apply>
```

The above rules apply to all symbols in classes `nary-reln.class` and `nary-set-reln.class`. In the latter case the choice of Content Dictionary to use depends on the `type` attribute on the symbol, defaulting to set1, but multiset1 should be used if `type=multiset`.

F.6.3 Apply to list

Into a construction with the apply_to_list symbol via the general rule Rewrite: n-ary domainofapplication for general n-ary operators.

If the argument list is given explicitly, the Rewrite: element rule applies.

Any use of qualifier elements is expressed in Strict Content MathML via explicitly applying the function to a list of arguments using the apply_to_list symbol as shown in the following rule. The rule only considers the `domainofapplication` qualifier as other qualifiers may be rewritten to `domainofapplication` as described earlier.

Rewrite: n-ary domainofapplication

An expression of the following form, where `<union/>` represents any element of the relevant class and `<ci>expression-in-x</ci>` is an arbitrary expression involving the bound variable(s)

```
<apply><union/>
  <bvar><ci>x</ci></bvar>
  <domainofapplication><ci>D</ci></domainofapplication>
  <ci>expression-in-x</ci>
</apply>
```

is rewritten to

```
<apply><csymbol cd="fns2">apply_to_list</csymbol>
  <csymbol cd="set1">union</csymbol>
  <apply><csymbol cd="list1">map</csymbol>
    <bind><csymbol cd="fns1">lambda</csymbol>
      <bvar><ci>x</ci></bvar>
      <ci>expression-in-x</ci>
    </bind>
    <ci>D</ci>
  </apply>
</apply>
```

The above rule applies to all symbols in the listed classes. In the case of `nary-set.class` the choice of Content Dictionary to use depends on the `type` attribute on the arguments, defaulting to set1, but multiset1 should be used if `type=multiset`.

Note that the members of the `nary-constructor.class`, such as `vector`, use *constructor* syntax where the arguments and qualifiers are given as children of the element rather than as children of a containing `apply`. In this case, the above rules apply with the analogous syntactic modifications.

F.6.4 Such that

Into a construction using the suchthat symbol from the set1 content dictionary in an `apply` with bound variables via the Rewrite: apply bvar domainofapplication rule.

In general, an application involving bound variables and (possibly) `domainofapplication` is rewritten using the following rule, which makes the domain the first positional argument of the application, and uses the lambda symbol to encode the variable bindings. Certain classes of operator have alternative rules, as described below.

Rewrite: apply bvar domainofapplication

A content MathML expression with bound variables and `domainofapplication`

```
        <apply><ci>H</ci>
          <bvar><ci>v1</ci></bvar>
...
          <bvar><ci>vn</ci></bvar>
          <domainofapplication><ci>D</ci></domainofapplication>
          <ci>A1</ci>
...
          <ci>Am</ci>
        </apply>
```

is rewritten to

```
        <apply><ci>H</ci>
          <ci>D</ci>
          <bind><csymbol cd="fns1">lambda</csymbol>
            <bvar><ci>v1</ci></bvar>
...
            <bvar><ci>vn</ci></bvar>
            <ci>A1</ci>
          </bind>
...
          <bind><csymbol cd="fns1">lambda</csymbol>
            <bvar><ci>v1</ci></bvar>
...
            <bvar><ci>vn</ci></bvar>
            <ci>Am</ci>
          </bind>
        </apply>
```

If there is no `domainofapplication` qualifier the `<ci>D</ci>` child is omitted.

F.7 Rewrite token elements

Rewrite non-strict token elements

F.7.1 Numbers

Rewrite numbers represented as `cn` elements where the `type` attribute is one of `e-notation`, `rational`, `complex-cartesian`, `complex-polar`, `constant` as strict `cn` via rules Rewrite: cn sep, Rewrite: cn based_integer and Rewrite: cn constant.

Rewrite: cn sep

If there are `sep` children of the `cn`, then intervening text may be rewritten as `cn` elements. If the `cn` element containing

`sep` also has a `base` attribute, this is copied to each of the `cn` arguments of the resulting symbol, as shown below.

```
<cn type="rational" base="b">n<sep/>d</cn>
```

is rewritten to

```
<apply><csymbol cd="nums1">rational</csymbol>
  <cn type="integer" base="b">n</cn>
  <cn type="integer" base="b">d</cn>
</apply>
```

The symbol used in the result depends on the `type` attribute according to the following table:

type attribute	OpenMath Symbol
e-notation	bigfloat
rational	rational
complex-cartesian	complex_cartesian
complex-polar	complex_polar

Note: In the case of bigfloat the symbol takes three arguments, `<cn type="integer">10</cn>` should be inserted as the second argument, denoting the base of the exponent used.

If the `type` attribute has a different value, or if there is more than one `<sep/>` element, then the intervening expressions are converted as above, but a system-dependent choice of symbol for the head of the application must be used.

If a base attribute has been used then the resulting expression is not Strict Content MathML, and each of the arguments needs to be recursively processed.

Rewrite: cn based_integer

A `cn` element with a base attribute other than 10 is rewritten as follows. (A base attribute with value 10 is simply removed.)

```
<cn type="integer" base="16">FF60</cn>
```

```
<apply><csymbol cd="nums1">based_integer</csymbol>
  <cn type="integer">16</cn>
  <cs>FF60</cs>
</apply>
```

If the original element specified type `integer` or if there is no type attribute, but the content of the element just consists of the characters [a-zA-Z0-9] and white space then the symbol used as the head in the resulting application should be based_integer as shown. Otherwise it should be based_float.

Rewrite: cn constant

In Strict Content MathML, constants should be represented using `csymbol` elements. A number of important constants are defined in the nums1 content dictionary. An expression of the form

```
<cn type="constant">c</cn>
```

has the Strict Content MathML equivalent

```
<csymbol cd="nums1">c2</csymbol>
```

where `c2` corresponds to `c` as specified in the following table.

Content	Description	OpenMath Symbol
U+03C0 (`&pi;`)	The usual π of trigonometry: approximately 3.141592653...	pi
U+2147 (`&ExponentialE;` or `&ee;`)	The base for natural logarithms: approximately 2.718281828...	e
U+2148 (`&ImaginaryI;` or `&ii;`)	Square root of -1	i
U+03B3 (`&gamma;`)	Euler's constant: approximately 0.5772156649...	gamma
U+221E (`&infin;` or `&infty;`)	Infinity. Proper interpretation varies with context	infinity

F.7.2 Token presentation

Rewrite any `ci`, `csymbol` or `cn` containing presentation MathML to `semantics` elements with rules Rewrite: cn presentation mathml and Rewrite: ci presentation mathml and the analogous rule for `csymbol`.

Rewrite: cn presentation mathml

If the `cn` contains Presentation MathML markup, then it may be rewritten to Strict MathML using variants of the rules above where the arguments of the constructor are `ci` elements annotated with the supplied Presentation MathML.

A `cn` expression with non-text content of the form

```
<cn type="rational"><mi>P</mi><sep/><mi>Q</mi></cn>
```

is transformed to Strict Content MathML by rewriting it to

```
<apply><csymbol cd="nums1">rational</csymbol>
  <semantics>
    <ci>p</ci>
    <annotation-xml encoding="MathML-Presentation">
      <mi>P</mi>
    </annotation-xml>
  </semantics>
  <semantics>
    <ci>q</ci>
    <annotation-xml encoding="MathML-Presentation">
      <mi>Q</mi>
    </annotation-xml>
  </semantics>
</apply>
```

Where the identifier names, p and q, (which have to be a text string) should be determined from the presentation MathML content, in a system defined way, perhaps as in the above example by taking the character data of the element ignoring any element markup. Systems doing such rewriting should ensure that constructs using the same Presentation MathML content are rewritten to `semantics` elements using the same `ci`, and that conversely constructs that use different MathML should be rewritten to different identifier names (even if the Presentation MathML has the same character data).

A related special case arises when a `cn` element contains character data not permitted in Strict Content MathML usage, e.g. non-digit, alphabetic characters. Conceptually, this is analogous to a `cn` element containing a presentation markup `mtext` element, and could be rewritten accordingly. However, since the resulting annotation would contain no additional rendering information, such instances should be rewritten directly as `ci` elements, rather than as a `semantics` construct.

The `ci` element can contain `mglyph` elements to refer to characters not currently available in Unicode, or a general presentation construct (see 3.1.8 Summary of Presentation Elements), which is used for rendering (see 4.1.2 Content Expressions).

Rewrite: ci presentation mathml

A `ci` expression with non-text content of the form

```
<ci><mi>P</mi></ci>
```

is transformed to Strict Content MathML by rewriting it to

```
<semantics>
  <ci>p</ci>
  <annotation-xml encoding="MathML-Presentation">
    <mi>P</mi>
  </annotation-xml>
</semantics>
```

Where the identifier name, p, (which has to be a text string) should be determined from the presentation MathML content, in a system defined way, perhaps as in the above example by taking the character data of the element ignoring any element markup. Systems doing such rewriting should ensure that constructs using the same Presentation MathML

content are rewritten to `semantics` elements using the same `ci`, and that conversely constructs that use different MathML should be rewritten to different identifier names (even if the Presentation MathML has the same character data).

The following example encodes an atomic symbol that displays visually as C^2 and that, for purposes of content, is treated as a single symbol

```
<ci>
  <msup><mi>C</mi><mn>2</mn></msup>
</ci>
```

The Strict Content MathML equivalent is

```
<semantics>
  <ci>C2</ci>
  <annotation-xml encoding="MathML-Presentation">
    <msup><mi>C</mi><mn>2</mn></msup>
  </annotation-xml>
</semantics>
```

F.8 Rewrite operators

Rewrite any remaining operator defined in 4.3 Content MathML for Specific Structures to a `csymbol` referencing the symbol identified in the syntax table by the rule Rewrite: element.

Rewrite: element

For example,

```
<plus/>
```

is equivalent to the Strict form

```
<csymbol cd="arith1">plus</csymbol>
```

As noted in the descriptions of each operator element, some operators require special case rules to determine the proper choice of symbol. Some cases of particular note are:

1. The order of the arguments for the `selector` operator must be rewritten, and the symbol depends on the type of the arguments.
2. The choice of symbol for the `minus` operator depends on the number of the arguments, `minus` or `minus`.
3. The choice of symbol for some set operators depends on the values of the `type` of the arguments.

4. The choice of symbol for some statistical operators depends on the values of the types of the arguments.

5. The choice of symbol for the `emptyset` element depends on context.

F.8.1 Rewrite the `minus` operator

The `minus` element can be used as a *unary arithmetic operator* (e.g. to represent - *x*), or as a *binary arithmetic operator* (e.g. to represent *x*- *y*).

If it is used with one argument, `minus` corresponds to the unary_minus symbol.

If it is used with two arguments, `minus` corresponds to the minus symbol

In both cases, the translation to Strict Content markup is direct, as described in Rewrite: element. It is merely a matter of choosing the symbol that reflects the actual usage.

F.8.2 Rewrite the set operators

When translating to Strict Content Markup, if the `type` has value `multiset`, then the in symbol from multiset1 should be used instead.

When translating to Strict Content Markup, if the `type` has value `multiset`, then the notin symbol from multiset1 should be used instead.

When translating to Strict Content Markup, if the `type` has value `multiset`, then the subset symbol from multiset1 should be used instead.

When translating to Strict Content Markup, if the `type` has value `multiset`, then the prsubset symbol from multiset1 should be used instead.

When translating to Strict Content Markup, if the `type` has value `multiset`, then the notsubset symbol from multiset1 should be used instead.

When translating to Strict Content Markup, if the `type` has value `multiset`, then the notprsubset symbol from multiset1 should be used instead.

When translating to Strict Content Markup, if the `type` has value `multiset`, then the setdiff symbol from multiset1 should be used instead.

When translating to Strict Content Markup, if the `type` has value `multiset`, then the size symbol from multiset1 should be used instead.

F.8.3 Rewrite the statistical operators

When the `mean` element is applied to an explicit list of arguments, the translation to Strict Content markup is direct, using

the mean symbol from the s_data1 content dictionary, as described in Rewrite: element. When it is applied to a distribution, then the mean symbol from the s_dist1 content dictionary should be used. In the case with qualifiers use Rewrite: n-ary domainofapplication with the same caveat.

When the `sdev` element is applied to an explicit list of arguments, the translation to Strict Content markup is direct, using the sdev symbol from the s_data1 content dictionary, as described in Rewrite: element. When it is applied to a distribution, then the sdev symbol from the s_dist1 content dictionary should be used. In the case with qualifiers use Rewrite: n-ary domainofapplication with the same caveat.

When the `variance` element is applied to an explicit list of arguments, the translation to Strict Content markup is direct, using the variance symbol from the s_data1 content dictionary, as described in Rewrite: element. When it is applied to a distribution, then the variance symbol from the s_dist1 content dictionary should be used. In the case with qualifiers use Rewrite: n-ary domainofapplication with the same caveat.

When the `median` element is applied to an explicit list of arguments, the translation to Strict Content markup is direct, using the median symbol from the s_data1 content dictionary, as described in Rewrite: element.

When the `mode` element is applied to an explicit list of arguments, the translation to Strict Content markup is direct, using the mode symbol from the s_data1 content dictionary, as described in Rewrite: element.

F.8.4 Rewrite the `emptyset` operator

In some situations, it may be clear from context that `emptyset` corresponds to the emptyset symbol from the multiset1 content dictionary. However, as there is no method other than annotation for an author to explicitly indicate this, it is always acceptable to translate to the emptyset symbol from the set1 content dictionary.

F.9 Rewrite attributes

F.9.1 Rewrite the `type` attribute

At this point, all elements that accept the `type`, other than `ci` and `csymbol`, should have been rewritten into Strict Content Markup equivalents without `type` attributes, where type information is reflected in the choice of operator symbol. Now rewrite remaining `ci` and `csymbol` elements with a `type` attribute to a strict expression with `semantics` according to rules Rewrite: ci type annotation and Rewrite: csymbol type annotation.

Rewrite: ci type annotation

In Strict Content, type attributes are represented via semantic attribution. An expression of the form

```
<ci type="T">n</ci>
```

is rewritten to

```
<semantics>
  <ci>n</ci>
  <annotation-xml cd="mathmltypes" name="type" encoding="MathML-Content">
    <ci>T</ci>
  </annotation-xml>
</semantics>
```

In non-Strict usage `csymbol` allows the use of a `type` attribute.

Rewrite: csymbol type annotation

In Strict Content, type attributes are represented via semantic attribution. An expression of the form

```
<csymbol type="T">symbolname</csymbol>
```

is rewritten to

```
<semantics>
  <csymbol>symbolname</csymbol>
  <annotation-xml cd="mathmltypes" name="type" encoding="MathML-Content">
    <ci>T</ci>
  </annotation-xml>
</semantics>
```

F.9.2 Rewrite `definitionURL` and `encoding` attributes

If the `definitionURL` and `encoding` attributes on a `csymbol` element can be interpreted as a reference to a content dictionary (see 4.2.3.2 Non-Strict uses of `<csymbol>` for details), then rewrite to reference the content dictionary by the `cd` attribute instead.

F.9.3 Rewrite attributes

Rewrite any element with attributes that are not allowed in strict markup to a `semantics` construction with the element without these attributes as the first child and the attributes in `annotation` elements by rule Rewrite: attributes.

A number of content MathML elements such as `cn` and `interval` allow attributes to specialize the semantics of the objects they represent. For these cases, special rewrite rules are given on a case-by-case basis in 4.3 Content MathML for Specific Structures. However, content MathML elements also accept attributes shared by all MathML elements, and depending on the context, may also contain attributes from other XML namespaces. Such attributes must be rewritten in alternative form in Strict Content Markup.

Rewrite: attributes

For instance,

```
<ci class="foo" xmlns:other="http://example.com" other:att="bla">x</ci>
```

is rewritten to

```
 <semantics>
   <ci>x</ci>
   <annotation cd="mathmlattr"
name="class" encoding="text/plain">foo</annotation>
     <annotation-xml cd="mathmlattr" name="foreign" encoding="MathML-Content">
       <apply><csymbol cd="mathmlattr">foreign_attribute</csymbol>
         <cs>http://example.com</cs>
         <cs>other</cs>
         <cs>att</cs>
         <cs>bla</cs>
       </apply>
     </annotation-xml>
   </semantics>
```

For MathML attributes not allowed in Strict Content MathML the content dictionary mathmlattr is referenced, which provides symbols for all attributes allowed on content MathML elements.

G. MathML Index

G.1 Index of elements

variance
4.3.5.13 N-ary/Unary Statistical Operators: <mean/>, <median/>, <mode/>, <sdev/>, <variance/> F.8.3 Rewrite the statistical operators

vector
4.2.2.1 Strict uses of <ci> 4.3.5.8 N-ary Matrix Constructors: <vector/>, <matrix/>, <matrixrow/> F.6.3 Apply to list

vectorproduct
4.3.6.4 Binary Linear Algebra Operators: <vectorproduct/>, <scalarproduct/>, <outerproduct/>

xor
4.3.5.5 N-ary Logical Operators: <and/>, <or/>, <xor/>

H. Working Group Membership and Acknowledgments

This section is non-normative.

H.1 The Math Working Group Membership

The current Math Working Group is chartered from April 2021, until May 2023 and is co-chaired by Neil Soiffer and Brian Kardell (Igalia).

Between 2019 and 2021 the W3C MathML-Refresh Community Group was chaired by Neil Soiffer and developed the initial proposal for MathML Core and requirements for MathML 4.

The W3C Math Working Group responsible for MathML 3 (2012–2013) was co-chaired by David Carlisle of NAG and Patrick Ion of the AMS; Patrick Ion and Robert Miner of Design Science were co-chairs 2006-2011. Contact the co-chairs about membership in the Working Group. For the current membership see the W3C Math home page.

Robert Miner, whose leadership and contributions were essential to the development of the Math Working Group and MathML from their beginnings, died tragically young in December 2011.

Participants in the Working Group responsible for MathML 3.0 have been:

> Ron Ausbrooks, Laurent Bernardin, Pierre-Yves Bertholet, Bert Bos, Mike Brenner, Olga Caprotti, David Carlisle, Giorgi Chavchanidze, Ananth Coorg, Stéphane Dalmas, Stan Devitt, Sam Dooley, Margaret Hinchcliffe, Patrick Ion, Michael Kohlhase, Azzeddine Lazrek, Dennis Leas, Paul Libbrecht, Manolis Mavrikis, Bruce Miller, Robert Miner, Chris Rowley, Murray Sargent III, Kyle Siegrist, Andrew Smith, Neil Soiffer, Stephen Watt, Mohamed Zergaoui

All the above persons have been members of the Math Working Group, but some not for the whole life of the Working Group. The 22 authors listed for MathML3 at the start of that specification are those who contributed reworkings and reformulations used in the actual text of the specification. Thus the list includes the principal authors of MathML2 much of whose text was repurposed here. They were, of course, supported and encouraged by the activity and discussions of the whole Math Working Group, and by helpful commentary from outside it, both within the W3C and further afield.

For 2003 to 2006 W3C Math Activity comprised a Math Interest Group, chaired by David Carlisle of NAG and Robert Miner of Design Science.

The W3C Math Working Group (2001–2003) was co-chaired by Patrick Ion of the AMS, and Angel Diaz of IBM from June 2001 to May 2002; afterwards Patrick Ion continued as chair until the end of the WG's extended charter.

Participants in the Working Group responsible for MathML 2.0, second edition were:

> Ron Ausbrooks, Laurent Bernardin, Stephen Buswell, David Carlisle, Stéphane Dalmas, Stan Devitt, Max Froumentin, Patrick Ion, Michael Kohlhase, Robert Miner, Luca Padovani, Ivor Philips, Murray Sargent III, Neil Soiffer, Paul Topping, Stephen Watt

Earlier active participants of the W3C Math Working Group (2001 – 2003) have included:

> Angel Diaz, Sam Dooley, Barry MacKichan

The W3C Math Working Group was co-chaired by Patrick Ion of the AMS, and Angel Diaz of IBM from July 1998 to December 2000.

Participants in the Working Group responsible for MathML 2.0 were:

> Ron Ausbrooks, Laurent Bernardin, Stephen Buswell, David Carlisle, Stéphane Dalmas, Stan Devitt, Angel Diaz, Ben Hinkle, Stephen Hunt, Douglas Lovell, Patrick Ion, Robert Miner, Ivor Philips, Nico Poppelier, Dave Raggett, T.V. Raman, Murray Sargent III, Neil Soiffer, Irene Schena, Paul Topping, Stephen Watt

Earlier active participants of this second W3C Math Working Group have included:

> Sam Dooley, Robert Sutor, Barry MacKichan

At the time of release of MathML 1.0 [MathML1] the Math Working Group was co-chaired by Patrick Ion and Robert Miner, then of the Geometry Center. Since that time several changes in membership have taken place. In the course of the update to MathML 1.01, in addition to people listed in the original membership below, corrections were offered by David Carlisle, Don Gignac, Kostya Serebriany, Ben Hinkle, Sebastian Rahtz, Sam Dooley and others.

Participants in the Math Working Group responsible for the finished MathML 1.0 specification were:

> Stephen Buswell, Stéphane Dalmas, Stan Devitt, Angel Diaz, Brenda Hunt, Stephen Hunt, Patrick Ion, Robert Miner, Nico Poppelier, Dave Raggett, T.V. Raman, Bruce Smith, Neil Soiffer, Robert Sutor, Paul Topping, Stephen Watt, Ralph Youngen

Others who had been members of the W3C Math WG for periods at earlier stages were:

> Stephen Glim, Arnaud Le Hors, Ron Whitney, Lauren Wood, Ka-Ping Yee

H.2 Acknowledgments

The Working Group benefited from the help of many other people in developing the specification for MathML 1.0. We would like to particularly name Barbara Beeton, Chris Hamlin, John Jenkins, Ira Polans, Arthur Smith, Robby Villegas and Joe Yurvati for help and information in assembling the character tables in 8. Characters, Entities and Fonts, as well as Peter

Flynn, Russell S.S. O'Connor, Andreas Strotmann, and other contributors to the www-math mailing list for their careful proofreading and constructive criticisms.

As the Math Working Group went on to MathML 2.0, it again was helped by many from the W3C family of Working Groups with whom we necessarily had a great deal of interaction. Outside the W3C, a particularly active relevant front was the interface with the Unicode Technical Committee (UTC) and the NTSC WG2 dealing with ISO 10646. There the STIX project put together a proposal for the addition of characters for mathematical notation to Unicode, and this work was again spearheaded by Barbara Beeton of the AMS. The whole problem ended split into three proposals, two of which were advanced by Murray Sargent of Microsoft, a Math WG member and member of the UTC. But the mathematical community should be grateful for essential help and guidance over a couple of years of refinement of the proposals to help mathematics provided by Kenneth Whistler of Sybase, and a UTC and WG2 member, and by Asmus Freytag, also involved in the UTC and WG2 deliberations, and always a stalwart and knowledgeable supporter of the needs of scientific notation.

I. Changes

I.1 Changes between MathML 3.0 Second Edition and MathML 4.0

Changes to the Frontmatter

- Changes to the references to match new W3C specification rules, and to use the new W3C CSS formatting style, most notably affecting the table of contents styling.
- Update the Status of This Document, in particular using `https` and referencing the GitHub Issues page as required for current W3C publications.

Changes to 2. MathML Fundamentals

- Modified the definition of MathML color and length valued attributes to be explicitly based on the syntax used in [MathML-Core] which in turn uses definitions provided by CSS3.
- Remove the `mode` and `macros` attributes from `<math>`. These have been deprecated since MathML 2. `macros` had no defined behaviour, and `mode` can be replaced by suitable use of `display`. The `mathml4-legacy` schema makes these valid if needed for legacy applications.
- Remove the `other` attribute. This have been deprecated since MathML 2. The `mathml4-legacy` schema makes this valid if needed for legacy applications.

Changes to 3. Presentation Markup

- Separate the examples in 3.2.3.3 Examples and 3.2.4.3 Examples to improve their appearance when rendered.

- Clarify that negative numbers should be marked up with an explicit `mo` operator in 3.2.4.4 Numbers that should *not* be written using `<mn>` alone.
- Correct the long division notation names in 3.6.2.2 Attributes.
- Clarify that the horizontal alignment of scripts in 3.4.7 Prescripts and Tensor Indices `<mmultiscripts>`, `<mprescripts/>` is towards the base, and add a new example.
- The deprecated MathML 1 attributes on token elements: `fontfamily`, `fontweight`, `fontstyle`, `fontsize`, `color` and `background` are removed in favor of `mathvariant`, `mathsize`, `mathcolor` and `mathbackground`. These attributes are also no longer valid on `mstyle`. The `mathml4-legacy` schema makes these valid if needed for legacy applications.
- All the deprecated font related attributes have been dropped from `mglyph` which is still retained to include images in MathML.
- The value `indentingnewline` is no longer valid for `mspace` (it was equivalent to `newline`).
- In MathML table rows and cells must be explicitly marked wih `mtr` and `mtd`. The [MathML1] required that an implementation infer the row markup if it was omitted.
- The use of `malignmark` has been restricted and simplified, matching the features implemented in existing implementations. The `groupalign` attribute on table elements is no longer supported.
- The deprecated `mo` attributes, `fence` and `separator`, have been removed (and are also no longer listed as properties in B. Operator Dictionary). They are still valid in the A.2.6 Legacy MathML schema, but invalid in the default schema.
- The deprecated element `none` is replaced by an empty `mrow` throughout, to match [MathML-Core].
- The element `mlabeledtr` and the associated attributes `side` and `minlabelspacing` are no longer specified. They are removed from the default schema but valid in the Legacy Schema.
- To align with MathML-Core, the special extended syntax for `mpadded` length attributes `( "+" | "-" )? unsigned-number ( ("%" pseudo-unit?) | pseudo-unit | unit | namedspace )?` is no longer supported. Most of the functionality is still available using standard CSS length syntax. See Note: mpadded lengths.

Changes to 4. Content Markup

- Correct examples in 4.2.7.1 The `share` element, 4.2.7.2 An Acyclicity Constraint, 4.2.7.3 Structure Sharing and Binding, 4.2.7.4 Rendering Expressions with Structure Sharing to use `src` (to match the normative schema) not `href`. The previous examples were also valid, as `href` is a common presentational attribute allowed on all elements.
- The Content element syntax tables and mapping to OpenMath have been moved from Chapter 4 to a new appendix, E.3 The Content MathML Operators.
- All Information relating to the rewrite to Strict Content MathML has been collected together and moved to a new appendix F. The Strict Content MathML Transformation.
- The deprecated elements `reln`, `declare` and `fn` have been removed. The `mathml4-legacy` schema makes these valid if needed for legacy applications.

Changes to 5. Annotating MathML: intent

- Introduced new chapter 5. Annotating MathML: intent describing the `intent` attribute.

Changes to 6. Annotating MathML: semantics

- Renamed Chapter from “Mixing Markup Languages for Mathematical Expressions”
- The existing text on using the `<semantics>` element to mix Presentation and Content MathML is maintained in the second section, although reduced with some non normative text and examples moved to [MathML-Notes].
- MathML 3 deprecated the use of `encoding` and `definitionURL` on `<semantics>`. They are invalid in this specification. The `mathml4-legacy` schema may be used if these attributes need to be validated for a legacy application.

Changes to 7. Interactions with the Host Environment

- Some rewriting of the text and adjusting references as the Media type registrations have been moved from an Appendix of this specification to a separate document, [MathML-Media-Types].

Changes to Media Types

- Media type registrations have been moved from an Appendix of this specification to a separate document, [MathML-Media-Types].

Changes to A. Parsing MathML

- The schema was updated to match MathML4
- The schema was refactored with a new `mathml4-core` schema matching [MathML-Core] being used as the basis for `mathml4-presentation`, and a new `mathml4-legacy` schema that can be used to validate an existing corpus of documents matching [MathML3].

Changes to B. Operator Dictionary

- The spacing values and priorities of the elements were reviewed and adjusted.

- A new “compact” presentation is provided as well as the tabular presentation used previously.
- The underlying data files were updated to Unicode 14/15/16.

Changes to C. MathML Accessibility

- This new appendix collects together requirements and issues related to accessibility.

Changes to E.3 The Content MathML Operators and F. The Strict Content MathML Transformation

- These new appendices collect together the syntax tables, mappings to OpenMath and rewrite rules that were previously distributed throughout 4. Content Markup.

J. References

J.1 Normative references

[Bidi]
Unicode Bidirectional Algorithm. Manish Goregaokar मनीष गोरेगांवकर; Robin Leroy. Unicode Consortium. 2 September 2024. Unicode Standard Annex #9. URL: https://www.unicode.org/reports/tr9/tr9-50.html

[CSS-Color-3]
CSS Color Module Level 3. Tantek Çelik; Chris Lilley; David Baron. W3C. 18 January 2022. W3C Recommendation. URL: https://www.w3.org/TR/css-color-3/

[CSS-VALUES-3]
CSS Values and Units Module Level 3. Tab Atkins Jr.; Elika Etemad. W3C. 22 March 2024. W3C Candidate Recommendation. URL: https://www.w3.org/TR/css-values-3/

[CSS21]
Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. Bert Bos; Tantek Çelik; Ian Hickson; Håkon Wium Lie. W3C. 7 June 2011. W3C Recommendation. URL: https://www.w3.org/TR/CSS21/

[DLMF]
NIST Digital Library of Mathematical Functions, Release 1.1.5. F. W. J. Olver; A. B. Olde Daalhuis; D. W. Lozier; B. I. Schneider; R. F. Boisvert; C. W. Clark; B. R. Miller; B. V. Saunders; H. S. Cohl; M. A. McClain. 2022-03-15. URL: http://dlmf.nist.gov/

[Entities]
XML Entity Definitions for Characters (3rd Edition). Patrick D F Ion; David Carlisle. W3C. 7 March 2023. W3C Recommendation. URL: https://www.w3.org/TR/xml-entity-names/

[HTML]
HTML Standard. Anne van Kesteren; Domenic Denicola; Dominic Farolino; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[IEEE754]
IEEE754.

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

[IRI]
Internationalized Resource Identifiers (IRIs). M. Duerst; M. Suignard. IETF. January 2005. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc3987

[MathML-AAM]
MathML Accessiblity API Mappings 1.0. W3C. W3C Editor's Draft. URL: https://w3c.github.io/mathml-aam/

[MathML-Core]
MathML Core. David Carlisle; Frédéric Wang. W3C. 27 November 2023. W3C Working Draft. URL: https://www.w3.org/TR/mathml-core/

[MathML-Media-Types]
MathML Media-type Declarations. W3C. W3C Editor's Draft. URL: https://w3c.github.io/mathml-docs/mathml-media-types/

[Namespaces]
Namespaces in XML 1.0 (Third Edition). Tim Bray; Dave Hollander; Andrew Layman; Richard Tobin; Henry Thompson et al. W3C. 8 December 2009. W3C Recommendation. URL: https://www.w3.org/TR/xml-names/

[OpenMath]
The OpenMath Standard. S. Buswell; O. Caprotti; D. P. Carlisle; M. C. Dewar; M. Gaëtano; M. Kohlhase; J. H. Davenport; P. D. F. Ion; T. Wiesing. The OpenMath Society. July 2019. URL: https://openmath.org/standard/om20-2019-07-01/omstd20.html

[RELAXNG-SCHEMA]
Information technology -- Document Schema Definition Language (DSDL) -- Part 2: Regular-grammar-based validation -- RELAX NG. ISO/IEC. 2008. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c052348_ISO_IEC_19757-2_2008(E).zip

[RFC2045]
Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. N. Freed; N. Borenstein. IETF. November 1996. Draft Standard. URL: https://www.rfc-editor.org/rfc/rfc2045

[RFC2046]
Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. N. Freed; N. Borenstein. IETF. November 1996. Draft Standard. URL: https://www.rfc-editor.org/rfc/rfc2046

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc2119

[RFC3986]
Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee; R. Fielding; L. Masinter. IETF. January 2005. Internet Standard. URL: https://www.rfc-editor.org/rfc/rfc3986

[RFC7303]
XML Media Types. H. Thompson; C. Lilley. IETF. July 2014. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7303

[RFC8174]
Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc8174

[rfc9110]

HTTP Semantics. R. Fielding, Ed.; M. Nottingham, Ed.; J. Reschke, Ed.. IETF. June 2022. Internet Standard. URL: https://httpwg.org/specs/rfc9110.html

[SVG]
Scalable Vector Graphics (SVG) 1.1 (Second Edition). Erik Dahlström; Patrick Dengler; Anthony Grasso; Chris Lilley; Cameron McCormack; Doug Schepers; Jonathan Watt; Jon Ferraiolo; Jun Fujisawa; Dean Jackson et al. W3C. 16 August 2011. W3C Recommendation. URL: https://www.w3.org/TR/SVG11/

[UAAG20]
User Agent Accessibility Guidelines (UAAG) 2.0. James Allan; Greg Lowney; Kimberly Patch; Jeanne F Spellman. W3C. 15 December 2015. W3C Working Group Note. URL: https://www.w3.org/TR/UAAG20/

[Unicode]
The Unicode Standard. Unicode Consortium. URL: https://www.unicode.org/versions/latest/

[WCAG21]
Web Content Accessibility Guidelines (WCAG) 2.1. Michael Cooper; Andrew Kirkpatrick; Joshue O'Connor; Alastair Campbell. W3C. 21 September 2023. W3C Recommendation. URL: https://www.w3.org/TR/WCAG21/

[XML]
Extensible Markup Language (XML) 1.0 (Fifth Edition). Tim Bray; Jean Paoli; Michael Sperberg-McQueen; Eve Maler; François Yergeau et al. W3C. 26 November 2008. W3C Recommendation. URL: https://www.w3.org/TR/xml/

[XMLSchemaDatatypes]
XML Schema Part 2: Datatypes Second Edition. Paul V. Biron; Ashok Malhotra. W3C. 28 October 2004. W3C Recommendation. URL: https://www.w3.org/TR/xmlschema-2/

[XMLSchemas]
XML Schema Part 1: Structures Second Edition. Henry Thompson; David Beech; Murray Maloney; Noah Mendelsohn et al. W3C. 28 October 2004. W3C Recommendation. URL: https://www.w3.org/TR/xmlschema-1/

J.2 Informative references

[Concept-Lists]
Maintaining MathML Concept Lists. W3C. note. URL: https://w3c.github.io/mathml-docs/concept-lists/

[MathML-Notes]
Notes on MathML. W3C. note. URL: https://w3c.github.io/mathml-docs/notes-on-mathml/

[MathML-Types]
Structured Types in MathML 2.0. Stan Devitt; Michael Kohlhase; Max Froumentin. W3C. 10 November 2003. W3C Working Group Note. URL: https://www.w3.org/TR/mathml-types/

[MathML1]
Mathematical Markup Language (MathML) 1.0 Specification. Patrick D F Ion; Robert R Miner. W3C. 7 April 1998. W3C Recommendation. URL: https://www.w3.org/TR/1998/REC-MathML-19980407/

[MathML3]
Mathematical Markup Language (MathML) Version 3.0 2nd Edition. David Carlisle; Patrick D F Ion; Robert R Miner. W3C. 10 April 2014. W3C Recommendation. URL: https://www.w3.org/TR/MathML3/

[MathMLforCSS]
MathMLforCSS.

[Modularization]
XHTML™ Modularization 1.1. Daniel Austin; Subramanian Peruvemba; Shane McCarron; Masayasu Ishikawa; Mark Birbeck et al. W3C. 8 October 2008. W3C Recommendation. URL: https://www.w3.org/TR/2008/REC-xhtml-

modularization-20081008/

[OMDoc1.2]
OMDoc1.2.

[RDF]
Resource Description Framework (RDF): Concepts and Abstract Syntax. Graham Klyne; Jeremy Carroll. W3C. 10 February 2004. W3C Recommendation. URL: https://www.w3.org/TR/rdf-concepts/

[XHTML]
XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition). Steven Pemberton. W3C. 27 March 2018. W3C Recommendation. URL: https://www.w3.org/TR/xhtml1/

[XHTML-MathML-SVG]
An XHTML + MathML + SVG Profile. Masayasu Ishikawa. W3C. 9 August 2002. W3C Working Draft. URL: https://www.w3.org/TR/XHTMLplusMathMLplusSVG/

[XLink]
XML Linking Language (XLink) Version 1.0. Steven DeRose; Eve Maler; David Orchard. W3C. 27 June 2001. W3C Recommendation. URL: https://www.w3.org/TR/xlink/

↥