

BS ISO/IEC 40314:2016



BSI Standards Publication

Information technology — Mathematical Markup Language (MathML) Version 3.0 2nd Edition

bsi.

...making excellence a habit.
AVA Documents™
Document Sharing Hub

National foreword

This British Standard is the UK implementation of ISO/IEC 40314:2016.

The UK participation in its preparation was entrusted to Technical Committee ICT/-/1, Information systems co-ordination.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2016.
Published by BSI Standards Limited 2016

ISBN 978 0 580 87547 2

ICS 35.240.30

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 March 2016.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------

**Information technology —
Mathematical Markup Language
(MathML) Version 3.0 2nd Edition**

*Technologies de l'information — Langage de marquage
mathématique (MathML) Version 3.0 2e édition*





COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2016, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

FOREWORD

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT), see the following URL: [Foreword — Supplementary information](#).

ISO/IEC 40314 was prepared by W3C and was adopted, under the PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

Contents

1	Introduction	9
1.1	Mathematics and its Notation	9
1.2	Origins and Goals	10
1.2.1	Design Goals of MathML	10
1.3	Overview	11
1.4	A First Example	11
2	MathML Fundamentals	14
2.1	MathML Syntax and Grammar	14
2.1.1	General Considerations	14
2.1.2	MathML and Namespaces	14
2.1.3	Children versus Arguments	15
2.1.4	MathML and Rendering	15
2.1.5	MathML Attribute Values	15
2.1.6	Attributes Shared by all MathML Elements	20
2.1.7	Collapsing Whitespace in Input	21
2.2	The Top-Level <math> Element	22
2.2.1	Attributes	22
2.2.2	Deprecated Attributes	24
2.3	Conformance	24
2.3.1	MathML Conformance	24
2.3.2	Handling of Errors	27
2.3.3	Attributes for unspecified data	27
3	Presentation Markup	28
3.1	Introduction	28
3.1.1	What Presentation Elements Represent	28
3.1.2	Terminology Used In This Chapter	29
3.1.3	Required Arguments	30
3.1.4	Elements with Special Behaviors	31
3.1.5	Directionality	32
3.1.6	Displaystyle and Scriptlevel	33
3.1.7	Linebreaking of Expressions	34
3.1.8	Warning about fine-tuning of presentation	35
3.1.9	Summary of Presentation Elements	37
3.1.10	Mathematics style attributes common to presentation elements	38
3.2	Token Elements	38
3.2.1	Token Element Content Characters, <mglyph/>	39
3.2.2	Mathematics style attributes common to token elements	41
3.2.3	Identifier <mi>	45

CONTENTS

5

3.2.4	Number <code><mn></code>	46
3.2.5	Operator, Fence, Separator or Accent <code><mo></code>	47
3.2.6	Text <code><mtext></code>	60
3.2.7	Space <code><mspace/></code>	62
3.2.8	String Literal <code><ms></code>	64
3.3	General Layout Schemata	64
3.3.1	Horizontally Group Sub-Expressions <code><mrow></code>	64
3.3.2	Fractions <code><mfrac></code>	67
3.3.3	Radicals <code><msqrt></code> , <code><mroot></code>	69
3.3.4	Style Change <code><mstyle></code>	69
3.3.5	Error Message <code><merror></code>	72
3.3.6	Adjust Space Around Content <code><mpadded></code>	73
3.3.7	Making Sub-Expressions Invisible <code><mphantom></code>	78
3.3.8	Expression Inside Pair of Fences <code><mfenced></code>	80
3.3.9	Enclose Expression Inside Notation <code><menclose></code>	83
3.4	Script and Limit Schemata	85
3.4.1	Subscript <code><msub></code>	86
3.4.2	Superscript <code><msup></code>	87
3.4.3	Subscript-superscript Pair <code><msubsup></code>	87
3.4.4	Underscript <code><munder></code>	88
3.4.5	Overscript <code><mover></code>	89
3.4.6	Underscript-overscript Pair <code><munderover></code>	91
3.4.7	Prescripts and Tensor Indices <code><mmultiscripts></code> , <code><mprescripts/></code> , <code><none/></code>	93
3.5	Tabular Math	95
3.5.1	Table or Matrix <code><mtable></code>	95
3.5.2	Row in Table or Matrix <code><mtr></code>	99
3.5.3	Labeled Row in Table or Matrix <code><mlabeledtr></code>	99
3.5.4	Entry in Table or Matrix <code><mtd></code>	101
3.5.5	Alignment Markers <code><maligngroup/></code> , <code><malignmark/></code>	101
3.6	Elementary Math	110
3.6.1	Stacks of Characters <code><mstack></code>	111
3.6.2	Long Division <code><mlongdiv></code>	113
3.6.3	Group Rows with Similiar Positions <code><msgroup></code>	114
3.6.4	Rows in Elementary Math <code><msrow></code>	115
3.6.5	Carries, Borrows, and Crossouts <code><mscarries></code>	115
3.6.6	A Single Carry <code><mscarry></code>	116
3.6.7	Horizontal Line <code><msline/></code>	117
3.6.8	Elementary Math Examples	118
3.7	Enlivening Expressions	124
3.7.1	Bind Action to Sub-Expression <code><maction></code>	124
3.8	Semantics and Presentation	126
4	Content Markup	127
4.1	Introduction	127
4.1.1	The Intent of Content Markup	127
4.1.2	The Structure and Scope of Content MathML Expressions	128
4.1.3	Strict Content MathML	128
4.1.4	Content Dictionaries	129
4.1.5	Content MathML Concepts	130
4.2	Content MathML Elements Encoding Expression Structure	131

4.2.1	Numbers <cn>	132
4.2.2	Content Identifiers <ci>	138
4.2.3	Content Symbols <csymbol>	140
4.2.4	String Literals <cs>	142
4.2.5	Function Application <apply>	143
4.2.6	Bindings and Bound Variables <bind> and <bvar>	146
4.2.7	Structure Sharing <share>	148
4.2.8	Attribution via semantics	150
4.2.9	Error Markup <error>	151
4.2.10	Encoded Bytes <cbytes>	152
4.3	Content MathML for Specific Structures	152
4.3.1	Container Markup	153
4.3.2	Bindings with <apply>	154
4.3.3	Qualifiers	156
4.3.4	Operator Classes	162
4.3.5	Non-strict Attributes	169
4.4	Content MathML for Specific Operators and Constants	170
4.4.1	Functions and Inverses	170
4.4.2	Arithmetic, Algebra and Logic	180
4.4.3	Relations	200
4.4.4	Calculus and Vector Calculus	205
4.4.5	Theory of Sets	224
4.4.6	Sequences and Series	233
4.4.7	Elementary classical functions	243
4.4.8	Statistics	247
4.4.9	Linear Algebra	253
4.4.10	Constant and Symbol Elements	260
4.5	Deprecated Content Elements	268
4.5.1	Declare <declare>	268
4.5.2	Relation <reln>	268
4.5.3	Relation <fn>	268
4.6	The Strict Content MathML Transformation	268
5	Mixing Markup Languages for Mathematical Expressions	272
5.1	Annotation Framework	272
5.1.1	Annotation elements	272
5.1.2	Annotation keys	273
5.1.3	Alternate representations	274
5.1.4	Content equivalents	275
5.1.5	Annotation references	276
5.2	Elements for Semantic Annotations	276
5.2.1	The <semantics> element	276
5.2.2	The <annotation> element	277
5.2.3	The <annotation-xml> element	278
5.3	Combining Presentation and Content Markup	281
5.3.1	Presentation Markup in Content Markup	281
5.3.2	Content Markup in Presentation Markup	282
5.4	Parallel Markup	282
5.4.1	Top-level Parallel Markup	282
5.4.2	Parallel Markup via Cross-References	283

6	Interactions with the Host Environment	286
6.1	Introduction	286
6.2	Invoking MathML Processors	286
6.2.1	Recognizing MathML in XML	286
6.2.2	Recognizing MathML in HTML	287
6.2.3	Resource Types for MathML Documents	287
6.2.4	Names of MathML Encodings	287
6.3	Transferring MathML	288
6.3.1	Basic Transfer Flavor Names and Contents	288
6.3.2	Recommended Behaviors when Transferring	289
6.3.3	Discussion	289
6.3.4	Examples	290
6.4	Combining MathML and Other Formats	292
6.4.1	Mixing MathML and XHTML	294
6.4.2	Mixing MathML and non-XML contexts	294
6.4.3	Mixing MathML and HTML	294
6.4.4	Linking	295
6.4.5	MathML and Graphical Markup	296
6.5	Using CSS with MathML	297
6.5.1	Order of processing attributes versus style sheets	298
7	Characters, Entities and Fonts	299
7.1	Introduction	299
7.2	Unicode Character Data	299
7.3	Entity Declarations	300
7.4	Special Characters Not in Unicode	300
7.5	Mathematical Alphanumeric Symbols	300
7.6	Non-Marking Characters	303
7.7	Anomalous Mathematical Characters	303
7.7.1	Keyboard Characters	303
7.7.2	Pseudo-scripts	304
7.7.3	Combining Characters	306
A	Parsing MathML	308
A.1	Use of MathML as Well-Formed XML	308
A.2	Using the RelaxNG Schema for MathML3	308
A.2.1	Full MathML	309
A.2.2	Elements Common to Presentation and Content MathML	309
A.2.3	The Grammar for Presentation MathML	311
A.2.4	The Grammar for Strict Content MathML3	323
A.2.5	The Grammar for Content MathML	325
A.2.6	MathML as a module in a RelaxNG Schema	332
A.3	Using the MathML DTD	333
A.3.1	Document Validation Issues	333
A.3.2	Attribute values in the MathML DTD	333
A.3.3	DOCTYPE declaration for MathML	334
A.4	Using the MathML XML Schema	334
A.4.1	Associating the MathML schema with MathML fragments	334
A.5	Parsing MathML in XHTML	334
A.6	Parsing MathML in HTML	334

B	Media Types Registrations	335
B.1	Selection of Media Types for MathML Instances	335
B.2	Media type for Generic MathML	336
B.3	Media type for Presentation MathML	337
B.4	Media type for Content MathML	338
C	Operator Dictionary (Non-Normative)	340
C.1	Indexing of the operator dictionary	340
C.2	Format of operator dictionary entries	340
C.3	Notes on <code>lspace</code> and <code>rspace</code> attributes	341
C.4	Operator dictionary entries	341
D	Glossary (Non-Normative)	379
E	Working Group Membership and Acknowledgments (Non-Normative)	383
E.1	The Math Working Group Membership	383
E.2	Acknowledgments	386
F	Changes (Non-Normative)	387
F.1	Changes between MathML 3.0 First Edition and Second Edition	387
F.2	Changes between MathML 2.0 Second Edition and MathML 3.0	390
G	Normative References	391
H	References (Non-Normative)	393
I	Index (Non-Normative)	395
I.1	MathML Elements	395
I.2	MathML Attributes	400

Chapter 1

Introduction

1.1 Mathematics and its Notation

A distinguishing feature of mathematics is the use of a complex and highly evolved system of two-dimensional symbolic notation. As J. R. Pierce writes in his book on communication theory, mathematics and its notation should not be viewed as one and the same thing [Pierce1961]. Mathematical ideas can exist independently of the notation that represents them. However, the relation between meaning and notation is subtle, and part of the power of mathematics to describe and analyze derives from its ability to represent and manipulate ideas in symbolic form. The challenge before a Mathematical Markup Language (MathML) in enabling mathematics on the World Wide Web is to capture both notation and content (that is, its meaning) in such a way that documents can utilize the highly evolved notation of written and printed mathematics as well as the new potential for interconnectivity in electronic media.

Mathematical notation evolves constantly as people continue to innovate in ways of approaching and expressing ideas. Even the common notation of arithmetic has gone through an amazing variety of styles, including many defunct ones advocated by leading mathematical figures of their day [Cajori1928]. Modern mathematical notation is the product of centuries of refinement, and the notational conventions for high-quality typesetting are quite complicated and subtle. For example, variables and letters which stand for numbers are usually typeset today in a special mathematical italic font subtly distinct from the usual text italic; this seems to have been introduced in Europe in the late sixteenth century. Spacing around symbols for operations such as $+$, $-$, $\times$ and $/$ is slightly different from that of text, to reflect conventions about operator precedence that have evolved over centuries. Entire books have been devoted to the conventions of mathematical typesetting, from the alignment of superscripts and subscripts, to rules for choosing parenthesis sizes, and on to specialized notational practices for subfields of mathematics. The manuals describing the nuances of present-day computer typesetting and composition systems can run to hundreds of pages.

Notational conventions in mathematics, and in printed text in general, guide the eye and make printed expressions much easier to read and understand. Though we usually take them for granted, we, as modern readers, rely on numerous conventions such as paragraphs, capital letters, font families and cases, and even the device of decimal-like numbering of sections such as is used in this document. Such notational conventions are perhaps even more important for electronic media, where one must contend with the difficulties of on-screen reading. Appropriate standards coupled with computers enable a broadening of access to mathematics beyond the world of print. The markup methods for mathematics in use just before the Web rose to prominence importantly included T_EX (also written TeX) [Knuth1986] and approaches based on SGML ([AAP-math], [Poppelier1992] and [ISO-12083]).

It is remarkable how widespread the current conventions of mathematical notation have become. The general two-dimensional layout, and most of the same symbols, are used in all modern mathematical communications, whether the participants are, say, European, writing left-to-right, or Middle-Eastern,

writing right-to-left. Of course, conventions for the symbols used, particularly those naming functions and variables, may tend to favor a local language and script. The largest variation from the most common is a form used in some Arabic-speaking communities which lays out the entire mathematical notation from right-to-left, roughly in mirror image of the European tradition.

However, there is more to putting mathematics on the Web than merely finding ways of displaying traditional mathematical notation in a Web browser. The Web represents a fundamental change in the underlying metaphor for knowledge storage, a change in which *interconnection* plays a central role. It has become important to find ways of communicating mathematics which facilitate automatic processing, searching and indexing, and reuse in other mathematical applications and contexts. With this advance in communication technology, there is an opportunity to expand our ability to represent, encode, and ultimately to communicate our mathematical insights and understanding with each other. We believe that MathML as specified below is an important step in developing mathematics on the Web.

1.2 Origins and Goals

1.2.1 Design Goals of MathML

MathML has been designed from the beginning with the following ultimate goals in mind.

MathML should ideally:

- Encode mathematical material suitable for all educational and scientific communication.
 - Encode both mathematical notation and mathematical meaning.
 - Facilitate conversion to and from other mathematical formats, both presentational and semantic. Output formats should include:
 - graphical displays
 - speech synthesizers
 - input for computer algebra systems
 - other mathematics typesetting languages, such as $\text{\TeX}$
 - plain text displays, e.g. VT100 emulators
 - international print media, including braille
- It is recognized that conversion to and from other notational systems or media may entail loss of information in the process.
- Allow the passing of information intended for specific renderers and applications.
 - Support efficient browsing of lengthy expressions.
 - Provide for extensibility.
 - Be well suited to templates and other common techniques for editing formulas.
 - Be legible to humans, and simple for software to generate and process.

No matter how successfully MathML achieves its goals as a markup language, it is clear that MathML is useful only if it is implemented well. The W3C Math Working Group has identified a short list of additional implementation goals. These goals attempt to describe concisely the minimal functionality MathML rendering and processing software should try to provide.

- MathML expressions in HTML (and XHTML) pages should render properly in popular Web browsers, in accordance with reader and author viewing preferences, and at the highest quality possible given the capabilities of the platform.
- HTML (and XHTML) documents containing MathML expressions should print properly and at high-quality printer resolutions.
- MathML expressions in Web pages should be able to react to user gestures, such those as with a mouse, and to coordinate communication with other applications through the browser.

- Mathematical expression editors and converters should be developed to facilitate the creation of Web pages containing MathML expressions.

The extent to which these goals are ultimately met depends on the cooperation and support of browser vendors and other developers. The W3C Math Working Group has continued to work with other working groups of the W3C, and outside the W3C, to ensure that the needs of the scientific community will be met. MathML 2 and its implementations showed considerable progress in this area over the situation that obtained at the time of the MathML 1.0 Recommendation (April 1998) [MathML1]. MathML3 and the developing Web are expected to allow much more.

1.3 Overview

MathML is a markup language for describing mathematics. It is usually expressed in XML syntax, although HTML and other syntaxes are possible. A special aspect of MathML is that there are two main strains of markup: Presentation markup, discussed in Chapter 3, is used to display mathematical expressions; and Content markup, discussed in Chapter 4, is used to convey mathematical meaning. Content markup is specified in particular detail. This specification makes use of an XML format called Content Dictionaries. This format has been developed by the OpenMath Society, [OpenMath2004] with the dictionaries being used by this specification involving joint development by the OpenMath Society and the W3C Math Working Group.

Fundamentals common to both strains of markup are covered in Chapter 2, while the means for combining these strains, as well as external markup, into single MathML objects are discussed in Chapter 5. How MathML interacts with applications is covered in Chapter 6. Finally, a discussion of special symbols, and issues regarding characters, entities and fonts, is given in Chapter 7.

1.4 A First Example

The quadratic formula provides a simple but instructive illustration of MathML markup.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

MathML offers two flavors of markup of this formula. The first is the style which emphasizes the actual presentation of a formula, the two-dimensional layout in which the symbols are arranged. An example of this type is given just below. The second flavor emphasizes the mathematical content and an example of it follows the first one.

```
<mrow>
  <mi>x</mi>
  <mo>=</mo>
  <mfrac>
    <mrow>
      <mrow>
        <mo>-</mo>
        <mi>b</mi>
      </mrow>
      <mo>&PlusMinus;</mo>
    <msqrt>
```

```

<mrow>
  <msup>
    <mi>b</mi>
    <mn>2</mn>
  </msup>
  <mo>-</mo>
  <mrow>
    <mn>4</mn>
    <mo>&InvisibleTimes;</mo>
    <mi>a</mi>
    <mo>&InvisibleTimes;</mo>
    <mi>c</mi>
  </mrow>
</mrow>
</msqrt>
</mrow>
<mrow>
  <mn>2</mn>
  <mo>&InvisibleTimes;</mo>
  <mi>a</mi>
</mrow>
</mfrac>
</mrow>

```

Consider the superscript 2 in this formula. It represents the squaring operation here, but the meaning of a superscript in other situations depends on the context. A letter with a superscript can be used to signify a particular component of a vector, or maybe the superscript just labels a different type of some structure. Similarly two letters written one just after the other could signify two variables multiplied together, as they do in the quadratic formula, or they could be two letters making up the name of a single variable. What is called Content Markup in MathML allows closer specification of the mathematical meaning of many common formulas. The quadratic formula given in this style of markup is as follows.

```

<apply>
  <eq/>
  <ci>x</ci>
  <apply>
    <divide/>
    <apply>
      <plus/>
      <apply>
        <minus/>
        <ci>b</ci>
      </apply>
    <apply>
      <root/>
      <apply>
        <minus/>
        <apply>
          <power/>
          <ci>b</ci>
          <cn>2</cn>

```

1.4. A First Example

13

```
</apply>
<apply>
  <times/>
  <cn>4</cn>
  <ci>a</ci>
  <ci>c</ci>
</apply>
</apply>
</apply>
<apply>
  <times/>
  <cn>2</cn>
  <ci>a</ci>
</apply>
</apply>
</apply>
```

Chapter 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 [RELAX-NG]. The RelaxNG Schema is normative, but a DTD (Document Type Definition) and an XML Schema [XMLSchemas] are provided for continuity (they were normative for MathML2). See Appendix A.

MathML’s character set consists of legal characters as specified by Unicode [Unicode], further restricted by the characters not allowed in XML. The use of Unicode characters for mathematics is discussed in Chapter 7.

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. When the `xmlns` attribute is used alone, it sets the default namespace for the element on which it appears, and for any child elements. For example:

```
<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 XHTML, 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, which always places `math` elements and its descendents in the MathML namespace (other than special rules described in Appendix A for invalid input, and for annotation-xml). 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

Most MathML elements act as ‘containers’; such an element’s children are not distinguished from each other except as individual members of the list of children. Commonly there is no limit imposed on the number of children an element may have. This is the case for most presentation elements and some content elements such as `set`. But many MathML elements require a specific number of children, or attach a particular meaning to children in certain positions. Such elements are best considered to represent constructors of mathematical objects, and hence thought of as functions of their children. Therefore children of such a MathML element will often be referred to as its *arguments* instead of merely as children. Examples of this can be found, say, in Section 3.1.3.

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 Section 3.1.3.1.

In the detailed discussions of element syntax given with each element throughout the MathML specification, the correspondence of children with arguments, 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 Section 3.1.3.

2.1.4 MathML and Rendering

MathML presentation elements only recommend (i.e., do not require) specific ways of rendering; this is in order to allow for medium-dependent rendering and for individual preferences of style.

Nevertheless, some parts of this specification describe these recommended visual rendering rules in detail; in those descriptions it is often assumed that the model of rendering used supports the concepts of a well-defined ‘current rendering environment’ which, in particular, specifies a ‘current font’, a ‘current display’ (for pixel size) and a ‘current baseline’. The ‘current font’ provides certain metric properties and an encoding of glyphs.

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.

Except when explicitly forbidden by the specification for an attribute, MathML attribute values may contain any legal characters specified by the XML recommendation. See Chapter 7 for further clarification.

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. We use below the notation beginning with U+ that is recommended by Unicode for referring to Unicode characters [see [Unicode], page xxviii].

Notation	What it matches
<i>decimal-digit</i>	a decimal digit from the range U+0030 to U+0039
<i>hexadecimal-digit</i>	a hexadecimal (base 16) digit from the ranges U+0030 to U+0039, U+0041 to U+0046 and U+0061 to U+0066
<i>unsigned-integer</i>	a string of <i>decimal-digits</i> , representing a non-negative integer
<i>positive-integer</i>	a string of <i>decimal-digits</i> , but not consisting solely of "0"s (U+0030), representing a positive integer
<i>integer</i>	an optional "-" (U+002D), followed by a string of <i>decimal digits</i> , and representing an integer
<i>unsigned-number</i>	a string of <i>decimal digits</i> with up to one decimal point (U+002E), representing a non-negative terminating decimal number (a type of rational number)
<i>number</i>	an optional prefix of "-" (U+002D), followed by an unsigned number, representing a terminating decimal number (a type of rational number)
<i>character</i>	a single non-whitespace character
<i>string</i>	an arbitrary, nonempty and finite, string of <i>characters</i>
<i>length</i>	a length, as explained below, Section 2.1.5.2
<i>unit</i>	a unit, typically used as part of a length, as explained below, Section 2.1.5.2
<i>namedlength</i>	a named length, as explained below, Section 2.1.5.2
<i>color</i>	a color, as explained below, Section 2.1.5.3
<i>id</i>	an identifier, unique within the document; must satisfy the NAME syntax of the XML recommendation [XML]
<i>idref</i>	an identifier referring to another element within the document; must satisfy the NAME syntax of the XML recommendation [XML]
<i>URI</i>	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].
<i>italicized word</i>	values as explained in the text for each attribute; see Section 2.1.5.4
"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 as 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
$f_1 f_2 \dots f_n$	one instance of each form f_i , in sequence, with no separating whitespace
$f_1, f_2, \dots, f_n$	one instance of each form f_i , in sequence, with separating whitespace characters (but no commas)
$f_1 f_2 \dots f_n$	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, Appendix A.

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. The syntax of a length is specified as

Type	Syntax
<i>length</i>	<i>number</i> <i>number unit</i> <i>namedspace</i>

There should be no space between the number and the unit of a length.

The possible *units* and *namedspaces*, along with their interpretations, are shown below. Note that although the units and their meanings are taken from CSS, the syntax of lengths is not identical. A few MathML elements have length attributes that accept additional keywords; these are termed pseudo-units and specified in the description of those particular elements; see, for instance, Section 3.3.6.

A trailing "%" represents a percent of a reference value; unless otherwise stated, the reference value is the default value. The default value, or how it is obtained, is listed in the table of attributes for each element along with the reference value when it differs from the default. (See also Section 2.1.5.4.) A number without a unit is interpreted as a multiple of the reference value. This form is primarily for backward compatibility and should be avoided, preferring explicit units for clarity.

In some cases, the range of acceptable values for a particular attribute may be restricted; implementations are free to round up or down to the closest allowable value.

The possible *units* in MathML are:

Unit	Description
em	an em (font-relative unit traditionally used for horizontal lengths)
ex	an ex (font-relative unit traditionally used for vertical lengths)
px	pixels, or size of a pixel in the current display
in	inches (1 inch = 2.54 centimeters)
cm	centimeters
mm	millimeters
pt	points (1 point = 1/72 inch)
pc	picas (1 pica = 12 points)
%	percentage of the default value

Some additional aspects of units are discussed further below, in Section 2.1.5.2.

The following constants, *namedspaces*, may also be used where a length is needed; they are typically used for spacing or padding between tokens. Recommended default values for these constants are shown; the actual spacing used is implementation specific.

<i>namedspace</i>	Recommended default
"veryverythinmathspace"	1/18em
"verythinmathspace"	2/18em
"thinmathspace"	3/18em
"mediummathspace"	4/18em
"thickmathspace"	5/18em
"verythickmathspace"	6/18em
"veryverythickmathspace"	7/18em
"negativeveryverythinmathspace"	-1/18em
"negativeverythinmathspace"	-2/18em
"negativethinmathspace"	-3/18em
"negativemediummathspace"	-4/18em
"negativethickmathspace"	-5/18em
"negativeverythickmathspace"	-6/18em
"negativeveryverythickmathspace"	-7/18em

Additional notes about units

Lengths are only used in MathML for presentation, and presentation will ultimately involve rendering in or on some medium. For visual media, the display context is assumed to have certain properties available to the rendering agent. A px corresponds to a pixel on the display, to the extent that is meaningful. The resolution of the display device will affect the correspondence of pixels to the units in, cm, mm, pt and pc.

Moreover, the display context will also provide a default for the font size; the parameters of this font determine the initial values used to interpret the units em and ex, and thus indirectly the sizes of namedspaces. Since these units track the display context, and in particular, the user's preferences for display, the relative units em and ex are generally to be preferred over absolute units such as px or cm.

Two additional aspects of relative units must be clarified, however. First, some elements such as Section 3.4 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.

If, and how, lengths might affect non-visual media is implementation specific.

2.1.5.3 Color Valued Attributes

The color, or background color, of presentation elements may be specified as a *color* using the following syntax:

Type	Syntax
<i>color</i>	<code>#RGB</code> <code>#RRGGBB</code> <i>html-color-name</i>

A color is specified either by '#' followed by hexadecimal values for the red, green, and blue components, with no intervening whitespace, or by an *html-color-name*. The color components can be either 1-digit or 2-digit, but must all have the same number of digits; the component ranges from 0 (component not present) to FF (component fully present). Note that, for example, by the digit-doubling rule specified under Colors in [CSS21] #123 is a short form for #112233.

Color values can also be specified as an *html-color-name*, one of the color-name keywords defined in [HTML4] ("aqua", "black", "blue", "fuchsia", "gray", "green", "lime", "maroon", "navy", "olive", "purple", "red", "silver", "teal", "white", and "yellow"). Note that the color name keywords are not case-sensitive, unlike most keywords in MathML attribute values, for compatibility with CSS and HTML.

When a *color* is applied to an element, it is the color in which the content of tokens is rendered. Additionally, when inherited from a surrounding element or from the environment in which the complete MathML expression is embedded, it controls the color of all other drawing due to MathML elements, including the lines or radical signs that can be drawn in rendering *mfrac*, *mtable*, or *msqrt*.

When used to specify a background color, the keyword "transparent" is also allowed. The recommended MathML visual rendering rules do not define the precise extent of the region whose background is affected by using the *background* attribute on an element, except that, when the element's content does not have negative dimensions and its drawing region is not overlapped by other drawing due to surrounding negative spacing, this region should lie behind all the drawing done to render the content of the element, but 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 *background* attribute is not defined by these rules.

2.1.5.4 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 Section 3.3.4, 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 (Section 3.3.4) 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 Section 3.2.2), 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 Section 3.2. 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 Appendix C. This is also discussed in Section 3.2.5.7 and examples are given in Section 3.2.5.2.

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
<code>id</code>	<i>id</i> Establishes a unique identifier associated with the element to support linking, cross-references and parallel markup. See <code>xref</code> and Section 5.4.	<i>none</i>
<code>xref</code>	<i>idref</i> References another element within the document. See <code>id</code> and Section 5.4.	<i>none</i>
<code>class</code>	<i>string</i> Associates the element with a set of style classes for use with [XSLT] and [CSS21]. Typically this would be a space separated sequence of words, but this is not specified by MathML. See Section 6.5 for discussion of the interaction of MathML and CSS.	<i>none</i>
<code>style</code>	<i>string</i> Associates style information with the element for use with [XSLT] and [CSS21]. This typically would be an inline CSS style, but this is not specified by MathML. See Section 6.5 for discussion of the interaction of MathML and CSS.	<i>none</i>
<code>href</code>	<i>URI</i> Can be used to establish the element as a hyperlink to the specified <i>URI</i> .	<i>none</i>

Note that MathML 2 had no direct support for linking, and instead followed the W3C Recommendation ‘XML Linking Language’ [XLink] in defining links using the `xlink:href` attribute. This has changed, and MathML 3 now uses an `href` attribute. However, particular compound document formats may

specify the use of XML linking with MathML elements, so user agents that support XML linking should continue to support the use of the `xlink:href` attribute with MathML 3 as well.

See also Section 3.2.2 for a list of MathML attributes which can be used on most presentation token elements.

The attribute `other`, is deprecated (Section 2.3.3) in favor of the use of attributes from other namespaces.

Name	values	default
<code>other</code>	<i>string</i>	<i>none</i>
DEPRECATED but in MathML 1.0.		

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>
```

is equivalent to `<mtext>Theorem 1:</mtext>` or `<mtext>Theorem 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 ` ` (U+00A0) 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>
```

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 $\langle\text{math}\rangle$ 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 $\langle\text{math}\rangle$ 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 Section 6.3.

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 Section 3.3.4, including the common attributes specified in Section 2.1.6. 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 Section 3.1.5 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 Section 3.1.10)

In addition to those attributes, the math element accepts:

Name	values	default
display	"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 $\text{display}=\text{"block"}$, displaystyle is initialized to "true", whereas when $\text{display}=\text{"inline"}$, displaystyle is initialized to "false"; in both cases scriptlevel is initialized to 0 (See Section 3.1.6). 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.		
maxwidth	<i>length</i>	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.		
overflow	"linebreak" "scroll" "elide" "truncate" "scale"	linebreak
specifies the preferred handling in cases where an expression is too long to fit in the allowed width. See the discussion below.		
altimg	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-width	<i>length</i>	width of altimg
specifies the width to display altimg , scaling the image if necessary; See altimg-height .		

2.2. The Top-Level $\langle math \rangle$ Element

23

Name	values	default
altimg-height	<i>length</i>	<i>height of altimg</i>
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-valign	<i>length</i> "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	<i>string</i>	<i>none</i>
provides a textual alternative as a fall-back for user agents that do not support embedded MathML or images.		
cdgroup	<i>URI</i>	<i>none</i>
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 Section 4.2.3. 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 Section 3.1.7 for further discussion.
"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.
"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.
"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.

Value	Meaning
"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.

2.2.2 Deprecated Attributes

The following attributes of `math` are deprecated:

Name	values	default
macros	URI *	<i>none</i>
	intended to provide a way of pointing to external macro definition files. Macros are not part of the MathML specification.	
mode	"display" "inline"	inline
	specified whether the enclosed MathML expression should be rendered in a display style or an inline style. This attribute is deprecated in favor of the <code>display</code> attribute.	

2.3 Conformance

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.

2.3.1 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 Chapter 3. 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.

2.3.1.1 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.

2.3.1.2 *Deprecated MathML 1.x and MathML 2.x Features*

MathML 3.0 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 3.0. 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.

2.3.1.3 *MathML Extension Mechanisms and Conformance*

MathML 3.0 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 3.0 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 3.0 extension mechanisms are encouraged to monitor relevant standards activity (e.g., Unicode, OpenMath, etc.) and to update documents as more standardized markup becomes available.

2.3.2 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.

2.3.3 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 [MathML1] 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 remains deprecated in MathML 3.0.

For example, in MathML 1.0, it was recommended that if additional information was used in a renderer-specific implementation for the `maction` element (Section 3.7.1), that information should be passed in using the `other` attribute:

```
<maction actiontype="highlight" other="color='#ff0000'"> expression </maction>
```

From MathML 2.0 onwards, a `color` attribute from another namespace would be used:

```
<body xmlns:my="http://www.example.com/MathML/extensions">
...
<maction actiontype="highlight" my: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.

Chapter 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.

3.1.1 What Presentation Elements Represent

Presentation elements correspond to the ‘constructors’ of traditional mathematical notation — that is, to the basic kinds of symbols and expression-building structures out of which any particular piece of traditional mathematical notation is built. Because of the importance of traditional visual notation, the descriptions of the notational constructs the elements represent are usually given here in visual terms. However, the elements are medium-independent in the sense that they have been designed to contain enough information for good spoken renderings as well. Some attributes of these elements may make sense only for visual media, but most attributes can be treated in an analogous way in audio as well (for example, by a correspondence between time duration and horizontal extent).

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. This specification describes suggested visual rendering rules in some detail, but a particular MathML renderer is free to use its own rules as long as its renderings are intelligible.

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>
```

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 usually rendered invisibly. For example, the entities `ⅆ`, `ⅇ`, and `ⅈ` denote notational symbols semantically distinct from visually identical letters used as simple variables. Likewise, the entities `⁢`, `⁡`, `⁣` and the character U+2064 (INVISIBLE PLUS) usually render invisibly but represent significant information. These entities have distinct spoken renderings, may influence visual linebreaking and spacing, and may effect the evaluation or meaning of particular expressions. Accordingly, authors should use these entities wherever they are applicable. For instance, the expression represented visually as ' $f(x)$ ' would usually be spoken in English as ' f of x ' rather than just ' $f x$ '. MathML conveys this meaning by using the `⁡` operator after the ' f ', which, in this case, can be aurally rendered as 'of'.

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

3.1.2 Terminology Used In This Chapter

It is strongly recommended that, before reading the present chapter, one read Section 2.1 on MathML syntax and grammar, which contains important information on MathML notations and conventions. In particular, in this chapter it is assumed that the reader has an understanding of basic XML terminology described in Section 2.1.3, and the attribute value notations and conventions described in Section 2.1.5.

The remainder of this section introduces MathML-specific terminology and conventions used in this chapter.

3.1.2.1 Types of presentation elements

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 (except for whitespace, which they ignore). 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. The primary MathML token element types are identifiers (e.g. variables or function names), numbers, and operators (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. Note that although a token element represents a single meaningful 'symbol' (name, number, label, mathematical symbol, etc.), such symbols may be comprised of more than one character. For example `sin` and `24` are represented by the single tokens `<mi>sin</mi>` and `<mn>24</mn>` respectively.

In traditional mathematical notation, expressions are recursively constructed out of smaller expressions, and ultimately out of single symbols, with the parts grouped and positioned using one of a small set of notational structures, which can be thought of as 'expression constructors'. In MathML, expressions are constructed in the same way, with the layout schemata playing the role of the expression constructors. The layout schemata specify the way in which sub-expressions are built into larger expressions. The terminology derives from the fact that each layout schema corresponds to a different way of 'laying out' its sub-expressions to form a larger expression in traditional mathematical typesetting.

3.1.2.2 Terminology for other classes of elements and their relationships

The terminology used in this chapter for special classes of elements, and for relationships between elements, is as follows: The *presentation elements* are the MathML elements defined in this chapter.

These elements are listed in Section 3.1.9. The *content elements* are the MathML elements defined in Chapter 4.

A MathML *expression* is a single instance of any of the presentation elements with the exception of the empty elements `none` or `mprescripts`, or is a single instance of any of the content elements which are allowed as content of presentation elements (described in Section 5.3.2). A *sub-expression* of an expression *E* is any MathML expression that is part of the content of *E*, whether *directly* or *indirectly*, i.e. whether it is a ‘child’ of *E* or not.

Since 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 Section 3.2.7 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`, `mpantom`, `menclase`, `mtd`, `mscopy`, 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,

```
<mtd>
</mtd>
```

is treated as if it were

```
<mtd>
  <mrow>
  </mrow>
</mtd>
```

and

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

is treated as if it were

```
<msqrt>
  <mrow>
    <mo> - </mo>
```

```

      <mn> 1 </mn>
    </mrow>
  </msqrt>

```

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)
<code>mrow</code>	0 or more	
<code>mfrac</code>	2	<i>numerator denominator</i>
<code>msqrt</code>	1*	
<code>mroot</code>	2	<i>base index</i>
<code>mstyle</code>	1*	
<code>merror</code>	1*	
<code>mpadded</code>	1*	
<code>mphantom</code>	1*	
<code>mfenced</code>	0 or more	
<code>menclose</code>	1*	
<code>msub</code>	2	<i>base subscript</i>
<code>msup</code>	2	<i>base superscript</i>
<code>msubsup</code>	3	<i>base subscript superscript</i>
<code>munder</code>	2	<i>base underscript</i>
<code>mover</code>	2	<i>base overscript</i>
<code>munderover</code>	3	<i>base underscript overscript</i>
<code>mmultiscripts</code>	1 or more	<i>base (subscript superscript)* [<mprescripts/> (presubscript presuperscript)*]</i>
<code>mtable</code>	0 or more rows	0 or more <code>mtr</code> or <code>mlabeledtr</code> elements
<code>mlabeledtr</code>	1 or more	a label and 0 or more <code>mtd</code> elements
<code>mtr</code>	0 or more	0 or more <code>mtd</code> elements
<code>mtd</code>	1*	
<code>mstack</code>	0 or more	
<code>mlongdiv</code>	3 or more	<i>divisor result dividend (msrow msgroup mscarries msline)*</i>
<code>msgroup</code>	0 or more	
<code>msrow</code>	0 or more	
<code>mscarries</code>	0 or more	
<code>mscarry</code>	1*	
<code>maction</code>	1 or more	depend on <code>actiontype</code> attribute
<code>math</code>	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 Section 3.2.7. This definition affects some of the suggested rendering rules for `mo` elements (Section 3.2.5).

Certain elements, e.g. `msup`, are able to embellish operators that are their first argument. These elements are listed in Section 3.2.5, 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.

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 Section 2.2). 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 behaviour 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; ie. 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 individual subformula 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 used, such as RTL Arabic and LTR numbers, the Unicode bidirectional algorithm [Bidi] is 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 (Section 3.1.5.1) 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.

Please 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. These 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 `table`. 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 `scriptsize-multiplier` 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 `scriptsize-multiplier` is approximately the square root of 1/2 whereas `scriptminsize` defaults to 8 points; these values may be changed on `mstyle`; see Section 3.3.4. 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

Section 3.2.2) 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 Section 2.1.5.2) 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 Section 3.2.2), 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 Section 3.1.3.1) 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 render 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 a `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.

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

Automatic linebreaking of subexpressions of `mfrac`, `msqrt`, `mroot` and `menclase` 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 Section 3.2.5.2.)
- *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 Section 3.2.5.2.)
- *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 Section 3.2.5.2.)

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 Section 2.2.1.)

3.1.7.2 Automatic Linebreaking Algorithm (Informative)

One method of linebreaking that works reasonably well is sometimes referred to as a "best-fit" algorithm. It works by computing a "penalty" for each potential break point on a line. The break point with the smallest penalty is chosen and the algorithm then works on the next line. Three useful factors in a penalty calculation are:

1. How much of the line width (after subtracting of the indent) is unused? The more unused, the higher the penalty.
2. How deeply nested is the breakpoint in the expression tree? The expression tree's depth is roughly similar to the nesting depth of `mrows`. The more deeply nested the break point, the higher the penalty.
3. Does a linebreak here make layout of the next line difficult? If the next line is not the last line and if the `indentingstyle` uses information about the linebreak point to determine how much to indent, then the amount of room left for linebreaking on the next line must be considered; i.e., linebreaks that leave very little room to draw the next line result in a higher penalty.
4. Whether `"linebreak"` has been specified: `"nobreak"` effectively sets the penalty to infinity, `"badbreak"` increases the penalty `"goodbreak"` decreases the penalty, and `"newline"` effectively sets the penalty to 0.

This algorithm takes time proportional to the number of token elements times the number of lines.

3.1.7.3 Linebreaking Algorithm for Inline Expressions (Informative)

A common method for breaking inline expressions that are too long for the space remaining on the current line is to pick an appropriate break point for the expression and place the expression up to that point on the current line and place the remainder of the expression on the following line. This can be done by:

1. Querying the text processing engine for the minimum and maximum amount of space available on the current line.
2. Using a variation of the automatic linebreaking algorithm given above), and/or using hints provided by `linebreak` attributes on `mo` or `mspace` elements, to choose a line break. The goal is that the first part of the formula fits "comfortably" on the current line while breaking at a point that results in keeping related parts of an expression on the same line.
3. The remainder of the formula begins on the next line, positioned both vertically and horizontally according to the paragraph flow; MathML's `indentation` attributes are ignored in this algorithm.
4. If the remainder does not fit on a line, steps 1 - 3 are repeated for the second and subsequent lines. Unlike the for the first line, some part of the expression must be placed these lines so that the algorithm terminates.

3.1.8 Warning about fine-tuning of presentation

Some use-cases require precise control of the math layout and presentation. Several MathML elements and attributes expressly support such fine-tuning of the rendering. However, MathML rendering agents exhibit wide variability in their presentation of the the same MathML expression due to difference

in platforms, font availability, and requirements particular to the agent itself (see Section 3.1). The overuse of explicit rendering control may yield a ‘perfect’ layout on one platform, but give much worse presentation on others. The following sections clarify the kinds of problems that can occur.

3.1.8.1 Warning: non-portability of ‘tweaking’

For particular expressions, authors may be tempted to use the `mpadded`, `mspace`, `mphantom`, and `mtext` elements to improve (‘tweak’) the spacing generated by a specific renderer.

Without explicit spacing rules, various MathML renders may use different spacing algorithms. Consequently, different MathML renderers may position symbols in different locations relative to each other. Say that renderer B, for example, provides improved spacing for a particular expression over renderer A. Authors are strongly warned that ‘tweaking’ the layout for renderer A may produce very poor results in renderer B, very likely worse than without any explicit adjustment at all.

Even when a specific choice of renderer can be assumed, its spacing rules may be improved in successive versions, so that the effect of tweaking in a given MathML document may grow worse with time. Also, when style sheet mechanisms are extended to MathML, even one version of a renderer may use different spacing rules for users with different style sheets.

Therefore, it is suggested that MathML markup never use `mpadded` or `mspace` elements to tweak the rendering of specific expressions, unless the MathML is generated solely to be viewed using one specific version of one MathML renderer, using one specific style sheet (if style sheets are available in that renderer).

In cases where the temptation to improve spacing proves too strong, careful use of `mpadded`, `mphantom`, or the alignment elements (Section 3.5.5) may give more portable results than the direct insertion of extra space using `mspace` or `mtext`. Advice given to the implementers of MathML renderers might be still more productive, in the long run.

3.1.8.2 Warning: spacing should not be used to convey meaning

MathML elements that permit ‘negative spacing’, namely `mspace`, `mpadded`, and `mo`, could in theory be used to simulate new notations or ‘overstruck’ characters by the visual overlap of the renderings of more than one MathML sub-expression.

This practice is *strongly discouraged in all situations*, for the following reasons:

- it will give different results in different MathML renderers (so the warning about ‘tweaking’ applies), especially if attempts are made to render glyphs outside the bounding box of the MathML expression;
- it is likely to appear much worse than a more standard construct supported by good renderers;
- such expressions are almost certain to be uninterpretable by audio renderers, computer algebra systems, text searches for standard symbols, or other processors of MathML input.

More generally, any construct that uses spacing to convey mathematical meaning, rather than simply as an aid to viewing expression structure, is discouraged. That is, the constructs that are discouraged are those that would be interpreted differently by a human viewer of rendered MathML if all explicit spacing was removed.

Consider using the `mglyph` element for cases such as this. If such spacing constructs are used in spite of this warning, they should be enclosed in a `semantics` element that also provides an additional MathML expression that can be interpreted in a standard way. See Section 5.1 for further discussion.

The above warning also applies to most uses of rendering attributes to alter the meaning conveyed by an expression, with the exception of attributes on `mi` (such as `mathvariant`) used to distinguish one variable from another.

3.1.9 Summary of Presentation Elements

3.1.9.1 Token Elements

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

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

3.1.9.2 General Layout Schemata

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

3.1.9.3 Script and Limit Schemata

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

3.1.9.4 Tables and Matrices

<code>mtable</code>	table or matrix
<code>mlabeledtr</code>	row in a table or matrix with a label or equation number
<code>mtr</code>	row in a table or matrix
<code>mtd</code>	one entry in a table or matrix
<code>maligngroup</code> and <code>malignmark</code>	alignment markers

3.1.9.5 Elementary Math Layout

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

3.1.9.6 Enlivening Expressions

<code>maction</code>	bind actions to a sub-expression
----------------------	----------------------------------

3.1.10 Mathematics style attributes common to presentation elements

In addition to the attributes listed in Section 2.1.6, all MathML presentation elements accept the following two attributes:

Name	values	default
<code>mathcolor</code>	<i>color</i>	<i>inherited</i>
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 <code>mathcolor</code> used for child elements when used on a layout element.		
<code>mathbackground</code>	<i>color</i> "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.		

These style attributes 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` (See Section 3.2.2).

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 doesn't specify the mechanism by which style information is inherited from the rendering environment. See Section 3.2.2 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 is not overlapped by other drawing due to surrounding negative spacing, this region should lie behind all the drawing done to render the content, but 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.

Frequently, tokens consist of a single character denoting a mathematical symbol. Other cases, e.g. function names, involve multi-character tokens. Further, because traditional mathematical notation makes wide use of symbols distinguished by their typographical properties (e.g. a Fraktur 'g' for a Lie algebra, or a bold 'x' for a vector), care must be taken to insure that styling mechanisms respect typographical properties which carry meaning. Consequently, characters, tokens, and typographical properties of symbols are closely related to one another in MathML.

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/>`

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 Section 3.5.5).

Characters can be either represented directly as Unicode character data, or indirectly via numeric or character entity references. See Chapter 7 for a discussion of the advantages and disadvantages of numeric character references versus entity references, and [Entities] for a full list of the entity names available. Also, see Section 7.7 for a discussion of the appropriate character content to choose for certain applications.

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 `mglyphs` 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 Section 3.1.5.2).

3.2.1.1 Alphanumeric symbol characters

A large class of mathematical symbols are single letter identifiers typically used as variable names in formulas. Different font variants of a letter are treated as separate symbols. For example, a Fraktur 'g' might denote a Lie algebra, while a Roman 'g' denotes the corresponding Lie group. These letter-like symbols are traditionally typeset differently than the same characters appearing in text, using different spacing and ligature conventions. These characters must also be treated specially by style mechanisms, since arbitrary style transformations can change meaning in an expression.

For these reasons, Unicode contains more than nine hundred Math Alphanumeric Symbol characters corresponding to letter-like symbols. These characters are in the Secondary Multilingual Plane (SMP). See [Entities] for more information. As valid Unicode data, these characters are permitted in MathML and, as tools and fonts for them become widely available, we anticipate they will be the predominant way of denoting letter-like symbols.

MathML also provides an alternative encoding for these characters using only Basic Multilingual Plane (BMP) characters together with markup. MathML defines a correspondence between token elements

with certain combinations of BMP character data and the `mathvariant` attribute and tokens containing SMP Math Alphanumeric Symbol characters. Processing applications that accept SMP characters are required to treat the corresponding BMP and attribute combinations identically. This is particularly important for applications that support searching and/or equality testing.

The `mathvariant` attribute is described in more detail in Section 3.2.2, and a complete technical description of the corresponding characters is given in Section 7.5.

3.2.1.2 Using images to represent symbols `<mglyph/>`

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 `mglyphs` with the same values for `src` should be considered identical by applications that must determine whether two characters/glyphs are identical.

Attributes

The `mglyph` element accepts the attributes listed in Section 3.1.10, 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
<code>src</code>	<i>URI</i>	<i>required</i>
	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.	
<code>width</code>	<i>length</i>	<i>from image</i>
	Specifies the desired width of the glyph; see <code>height</code> .	
<code>height</code>	<i>length</i>	<i>from image</i>
	Specifies the desired height of the glyph. If only one of <code>width</code> and <code>height</code> 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.	
<code>valign</code>	<i>length</i>	<code>0ex</code>
	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.	
<code>alt</code>	<i>string</i>	<i>required</i>
	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.	

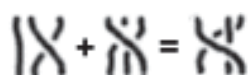
Note that the `src` and `alt` attributes are *required* for correct usage in MathML 3, however this is not enforced by the schema due to the deprecated usage described below.

Example

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:



Deprecated Attributes

Originally, `mglyph` was designed to provide access to non-standard fonts. Since this functionality was seldom implemented, nor were downloadable web fonts widely available, this use of `mglyph` has been deprecated. For reference, the following attributes were previously defined:

Name	values
<code>fontfamily</code>	<i>string</i> the name of a font that may be available to a MathML renderer, or a CSS font specification; See Section 6.5 and CSS [CSS21] for more information.
<code>index</code>	<i>integer</i> Specified a position of the desired glyph within the font named by the <code>fontfamily</code> attribute (see Section 3.2.2.1).

In MathML 1 and 2, both were required attributes; they are now optional and should be ignored unless the `src` attribute is missing.

Additionally, in MathML 2, `mglyph` accepted the attributes described in Section 3.2.2 (`mathvariant` and `mathsize`, along with the attributes deprecated there); to make clear that `mglyph` is *not* a token element, and since these attributes have no effect in any case, these attributes have been deprecated.

3.2.2 Mathematics style attributes common to token elements

In addition to the attributes defined for all presentation elements (Section 3.1.10), 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
<code>mathvariant</code>	"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 <code><mi></code>)
Specifies the logical class of the token. Note that this class is more than styling, it typically conveys semantic intent; see the discussion below.		
<code>mathsize</code>	"small" "normal" "big" <i>length</i>	<i>inherited</i>
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.		
<code>dir</code>	"ltr" "rtl"	<i>inherited</i>
specifies the initial directionality for text within the token: <code>ltr</code> (Left To Right) or <code>rtl</code> (Right To Left). This attribute should only be needed in rare cases involving weak or neutral characters; see Section 3.1.5.1 for further discussion. It has no effect on <code>mspace</code> .		

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 Section 6.5 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 to 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 Section 3.2.3 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 Alphanumeric 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

Section 7.5. 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 XHTML, 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 plugin, 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 Section 3.1.6.

3.2.2.1 *Deprecated style attributes on token elements*

The MathML 1.01 style attributes listed below are deprecated in MathML 2 and 3. These attributes were aligned to CSS but, in rendering environments that support CSS, it is preferable to use CSS directly to control the rendering properties corresponding to these attributes, rather than the attributes themselves. However as explained above, direct manipulation of these rendering properties by whatever means should usually be avoided. As a general rule, whenever there is a conflict between these deprecated attributes and the corresponding attributes (Section 3.2.2), the former attributes should be ignored.

The deprecated attributes are:

Name	values	default
fontfamily	<i>string</i>	<i>inherited</i>
Should be the name of a font that may be available to a MathML renderer, or a CSS font specification; See Section 6.5 and CSS [CSS21] for more information. Deprecated in favor of <i>mathvariant</i> .		
fontweight	"normal" "bold"	<i>inherited</i>
Specified the font weight for the token. Deprecated in favor of <i>mathvariant</i> .		
fontstyle	"normal" "italic"	normal (except on <mi>)
Specified the font style to use for the token. Deprecated in favor of <i>mathvariant</i> .		
fontsize	<i>length</i>	<i>inherited</i>
Specified the size for the token. Deprecated in favor of <i>mathsize</i> .		
color	<i>color</i>	<i>inherited</i>
Specified the color for the token. Deprecated in favor of <i>mathcolor</i> .		
background	<i>color</i> "transparent"	transparent
Specified the background color to be used to fill in the bounding box of the element and its children. Deprecated in favor of <i>mathbackground</i> .		

3.2.2.2 Embedding HTML in MathML

MathML can be combined with other formats as described in Section 6.4. 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 Section 3.2.6.4). The directionality, font size, and other font attributes should inherit from those that would be used for characters of the containing leaf element (see Section 3.2.2).

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

```
<math>
  <table>
    <tr>
      <td>
        <math><input type="text" placeholder="what shape is this?"></math>
      </td>
    </tr>
    <tr>
      <td>
        <math>
          <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:3"/>
          </svg>
        </math>
      </td>
    </tr>
  </table>
</math>
```

3.2.3 Identifier $\langle \text{mi} \rangle$

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 Section 3.2.1), 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) can be represented using an mi element, as shown in an example in Section 3.2.6.4.

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 Chapter 4.

3.2.3.2 Attributes

mi elements accept the attributes listed in Section 3.2.2, but in one case with a different default value:

Name	values	default
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"	(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 the deprecated fontstyle attribute defaults in the same way as mathvariant , depending on the content.

Note that for purposes of determining equivalences of Math Alphanumeric Symbol characters (See Section 7.5 and Section 3.2.1.1) the value of the mathvariant attribute should be resolved first, including the special defaulting behavior described above.

3.2.3.3 Examples

```
<mi> x </mi>
<mi> D </mi>
<mi> sin </mi>
<mi mathvariant='script'> L </mi>
<mi></mi>
```

An mi element with no content is allowed; $\langle \text{mi} \rangle \langle \text{mi} \rangle$ 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 (which also has the entity names `&af`; and `&ApplyFunction`;) as shown below; see also the discussion of invisible operators in Section 3.2.5.

```
<mrow>
  <mi> sin </mi>
  <mo> &ApplyFunction; </mo>
  <mi> x </mi>
</mrow>
```

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> &hellip; </mi>
  <mo> + </mo>
  <mi> n </mi>
</mrow>
```

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> &pi; </mi>
<mi> &ImaginaryI; </mi>
<mi> &ExponentialE; </mi>
```

3.2.4 Number `<mn>`

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 Section 3.2.1), 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 Chapter 4.

3.2.4.2 Attributes

`mn` elements accept the attributes listed in Section 3.2.2.

3.2.4.3 Examples

```

<mn> 2 </mn>
<mn> 0.123 </mn>
<mn> 1,000,000 </mn>
<mn> 2.1e10 </mn>
<mn> 0xFFEF </mn>
<mn> MCMLXIX </mn>
<mn> twenty one </mn>

```

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, ratios of numbers shown as fractions, and names of numeric constants. Examples of MathML representations of such numbers include:

```

<mrow>
  <mn> 2 </mn>
  <mo> + </mo>
  <mrow>
    <mn> 3 </mn>
    <mo> &InvisibleTimes; </mo>
    <mi> &ImaginaryI; </mi>
  </mrow>
</mrow>
<mfrac> <mn> 1 </mn> <mn> 2 </mn> </mfrac>
<mi> &pi; </mi>
<mi> &ExponentialE; </mi>

```

3.2.5 Operator, Fence, Separator or Accent <mo>

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 Section 3.2.1), 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> <= </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 `<=`.

All operators, in the general sense used here, are subject to essentially the same rendering attributes and rules. Subtle distinctions in the rendering of these classes of symbols, when they exist, are supported

using the Boolean attributes `fence`, `separator` and `accent`, which can be used to distinguish these cases.

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 values for `fence`, `separator` 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 Section 3.2.6. See also Chapter 4 for definitions of MathML content elements that are guaranteed to have the semantics of specific mathematical operators.

Note also that linebreaking, as discussed in Section 3.1.7, 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 the a linebreak is made at that operator.

3.2.5.2 Attributes

`mo` elements accept the attributes listed in Section 3.2.2 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 the Section 3.2.5.7. 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.