W3C Working Draft

Mathematical Markup Language (MathML) Version 2.0

W3C Working Draft 28 March 2000

This version: http://www.w3.org/TR/2000/WD-MathML2-20000328
Also available as: HTML zip archive, XHTML zip archive, XML zip archive, PDF (screen), PDF (paper)
Latest version: http://www.w3.org/TR/MathML2
Previous versions:
http://www.w3.org/TR/2000/WD-MathML2-20000211
http://www.w3.org/TR/1999/WD-MathML2-19991222
http://www.w3.org/TR/1999/WD-MathML2-19991201
Editors: Nico Poppelier (Penta Scope)
Robert Miner (Geometry Technologies, Inc.)
Patrick Ion (Mathematical Reviews, American Mathematical Society)
David Carlisle (NAG)
Principal Writers: Ron Ausbrooks, Stephen Buswell, Stéphane Dalmas, Stan Devitt, Angel Diaz, Roger Hunter, Bruce Smith, Neil Soiffer, Robert Sutor, Stephen Watt

Abstract

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

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

This document begins with background information on mathematical notation, the problems it poses, and the philosophy underlying the solutions MathML proposes. MathML can be used to encode both mathematical notation and mathematical content. About thirty of the MathML tags describe abstract notational structures, while another one hundred provide a way of unambiguously specifying the intended meaning of an expression. Additional chapters discuss how the MathML content and presentation elements interact, and how MathML renderers might be implemented and should interact with browsers. Finally, this document addresses the issue of MathML characters and their relation to fonts.

While MathML is human-readable, it is anticipated that, in all but the simplest cases, that authors will use equation editors, conversion programs, and other specialized software tools to generate MathML. Several early versions of such MathML tools already exist, and a number of others, both freely available software and commercial products, are under development.

Status of this document

This is the Last Call Working Draft of the MathML 2.0 specification prepared by the W3C Math Working Group. The Last Call review period ends 30 April 2000. The Math Working Group decided to proceed to Last Call with this draft at its recent Ann Arbor face-to-face meeting (minutes message).

This is a W3C Working Draft for review by W3C members and other interested parties. It is a draft document and may be updated, replaced or made obsolete by other documents at any time. It is inappropriate to use W3C working drafts as reference material or to cite them as other than ‘work in progress’. This is work in progress and does not imply endorsement by, or the consensus of, either W3C or participants in the Math Working Group.

This document has been produced by the W3C Math Working Group.

A list of current W3C Technical Reports can be found at http://www.w3.org/TR.

This document has been produced as part of the activity of the W3C User Interface Domain. The goals of the W3C Math Working Group are discussed in the W3C Math WG Charter (revised February 2000 from original of 11 June 1998). A list of participants in the W3C Math Working Group is available.

The present draft is a revision of the earlier corrected W3C Recommendation MathML 1.01. It differs from it in that all chapters have been updated and two added.

Chapters 1 and 2, which are introductory material have been revised to reflect the changes elsewhere in the document, and in the rapidly eveolving Web context. Chapters 3 and 4 have been extended to describe new functionalities added, as well as smaller improvements of material already proposed. Chapter 5 has been newly written to reflect changes in the technology available. The major tables in Chapter 6 have been regenerated to reflect an improved list of Unicode characters useful for mathematics, and the text revised to reflect the new preferred form for accessing them. Chapter 7 has been completely revised for Web technology has changed. A new chapter 8 on the DOM for MathML has been added; the latter points to a new appendix E for a detailed listing.

The appendices have been reorganized into normative and non-normative groups. Appendices E and H are completely new.

Comments on this document should be sent to the public mailing list of the Math Working Group.

Contents

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 notations. As J.R. Pierce has written in his book on communication theory, mathematics and its notations should not be viewed as one and the same thing [Pierce1961]. Mathematical ideas exist independently of the notations that represent 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 in putting mathematics on the World Wide Web is to capture both notation and content (that is, meaning) in such a way that documents can utilize the highly-evolved notational forms of written and printed mathematics, and the potential for interconnectivity in electronic media.

Mathematical notations are constantly evolving as people continue to make innovations in ways of approaching and expressing ideas. Even the commonplace notations of arithmetic have 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. For example, variables, or letters which stand for numbers, are usually typeset today in a special italic font subtly distinct from the usual text italic. Spacing around symbols for operations such as +, -, $\times$ and / is slightly different from that of text, to reflect conventions about operator precedence. Entire books have been devoted to the conventions of mathematical typesetting, from the alignment of superscripts and subscripts, to rules for choosing parenthesis sizes, to specialized notational practices for subfields of mathematics (for instance, [Chaundy1954], [Swanson1979],[Swanson1999], [Higham1993], or in the TEX literature [Knuth1986] and [Spivak1986]).

Notational conventions in mathematics, and 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 rely on hundreds of conventions such as paragraphs, capital letters, font families and cases, and even the device of decimal-like numbering of sections such as we are using in this document (an invention due to G. Peano, who is probably better known for his axioms for the natural numbers). Such notational conventions are perhaps even more important for electronic media, where one must contend with the difficulties of on-screen reading.

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 *interconnectivity* plays a central role. It is becoming increasingly 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 is an important step in developing mathematics on the Web.

1.2 Origins and Goals

1.2.1 The History of MathML

The problem of encoding mathematics for computer processing or electronic communication is much older than the Web. The common practice among scientists before the Web was to write papers in some encoded form based on the ASCII character set, and e-mail them to each other. Several markup methods for mathematics, in particular TEX [Knuth1986], were already in wide use in 1992 just before the Web rose to prominence, [Poppelier1992].

Since its inception, the Web has demonstrated itself to be a very effective method of making information available to widely separated groups of individuals. However, even though the World Wide Web was initially conceived and implemented by scientists for scientists, the possibilities for including mathematical expressions in HTML has been very limited. At present, most mathematics on the Web consists of text with images of scientific notation (in GIF or JPEG format), which are difficult to read and to author, or of entire documents in PDF form.

The World Wide Web Consortium (W3C) recognized that lack of support for scientific communication was a serious problem. Dave Raggett included a proposal for HTML Math in the HTML 3.0 working draft in 1994. A panel discussion on mathematical markup was held at the WWW Conference in Darmstadt in April 1995. In November 1995, representatives from Wolfram Research presented a proposal for doing math in HTML to the W3C team. In May 1996, the Digital Library Initiative meeting in Champaign-Urbana played an important role in bringing together many interested parties. Following the meeting, an HTML Math Editorial Review Board was formed. In the intervening years, this group has grown, and was formally reconstituted as the W3C Math working group in March 1997.

The MathML proposal reflects the interests and expertise of a very diverse group. Many contributions to the development of MathML deserve special mention, some of which we touch on here. One such contribution concerns the question of accessibility, especially for the visually handicapped. T.V. Raman is particularly notable in this regard. Neil Soiffer and Bruce Smith from Wolfram Research shared their experience with the problems of representing mathematics in connection with the design of Mathematica 3.0; this expertise an important influence in the design of the presentation elements. Paul Topping from Design Science also contributed his expertise in mathematical formatting and editing. MathML has benefited from the participation of a number of working group members involved in other mathematical encoding efforts in the SGML and computer-algebra communities, including Stephen Buswell from Stilo Technologies, Nico Poppelier (then with Elsevier Science), Stéphane Dalmas from INRIA (Sophia Antipolis), Stan Devitt at first with Waterloo Maple, Angel Diaz and Robert S. Sutor from IBM, and Stephen M. Watt from the University of Western Ontario. In particular, MathML has been influenced by the OpenMath project, the work of the ISO 12083 working group, and Stilo Technologies' work on a 'semantic' mathematics DTD fragment. The American Mathematical Society has played a key role in the development of MathML. Among other things, it has provided two working group chairs: Ron Whitney led the group from May 1996 to March 1997, and Patrick Ion, who has co-chaired the group with Robert Miner from The Geometry Center, from March 1997 to the present.

1.2.2 Acknowledgments

The working group benefited from the help of many other people in developing the specification for MathML 1.0. We would like to particularly name Barbara Beeton, Chris Hamlin, John Jenkins, Ira Polans, Arthur Smith, Robby Villegas and Joe Yurvati for help and information in assembling the character tables in Chapter 6, as well as Peter Flynn, Russel S.S. O'Connor, Andreas Strotmann, and other contributors to the www-math mailing list for their careful proofreading and constructive criticisms.

As the Math Working Group went on to MathML 2.0 it again was helped by many from the W3C family of Working Groups with whom we necessarily had a great deal of interaction. Outside the W3C, a particularly active relevant front was the interface with the Unicode Technical Committee (UTC) and the NTSC WG2 dealing with ISO 10646. There the STIX project put together a proposal for the addition of characters for mathematical notation to Unicode, and this work was again spear-headed by Barbara Beeton of the AMS. The whole problem ended split into three proposals, two of which were advanced by Murray Sargent of Microsoft, a Math WG member and member of the UTC. But the mathematical community should be grateful for essential help and guidance over a couple of years of refinement of the proposals to help mathematics provided by Kenneth Whistler of Oracle, and a UTC and WG2 member. Asmus Freitag, also involved in the UTC and WG2 deliberations was also a stalwart supporter of the needs of scientific notation.

1.2.3 Limitations of HTML

The demand for effective means of electronic scientific communication remains high. Ever increasingly, researchers, scientists, engineers, educators, students and technicians find themselves working at dispersed locations and relying on electronic communication. At the same time, the image-based methods that are currently the predominant means of transmitting scientific notation over the Web are primitive and inadequate. Document quality is poor, authoring is difficult, and mathematical information contained in images is not available for searching, indexing, or reuse in other applications.

The most obvious problems with HTML for mathematical communication are of two types.

Display Problems. Consider the equation $2^{2^x} = 10$. This equation is sized to match the surrounding line in 14pt type on the system where it was authored. Of course, on other systems, or for other font sizes, the equation is too small or too large. A second point to observe is that the equation image was generated against a white background. Thus, if a reader or browser resets the page background to another color, the anti-aliasing in the image results in white 'halos'. Next, consider the equation $x = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$ which is an example with the equation's horizontal alignment axis above the tops of the lower-case letters in surrounding text. This equation has a descender which places the baseline for the equation at a point about a third of the way from the bottom of the image. One can pad the image like this: $x = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$, so that the centerline of the image and the baseline of the equation coincide, but this causes problems with the inter-line spacing, resulting in the equation becoming difficult to read. Moreover, center alignment of images is handled in slightly different ways by different browsers, making it impossible to guarantee proper alignment for different clients.

Image-based equations are generally harder to see, read and comprehend than the surrounding text in the browser window. Moreover, these problems become worse when the document is printed. The resolution of the equations as images will be around 70 dots per inch, while the surrounding text will typically be 300, 600 or more dots per inch. The disparity in quality is judged to be unacceptable by most people.

Encoding Problems. Consider trying to search this document for part of an equation, for example, the '=10' from the first equation above. In a similar vein, consider trying to cut and paste an equation into another application; even more demanding is to cut and paste a sub-expression. Using image-based methods, neither of these common needs can be adequately addressed. Although the use of the `alt` in the document source can help, it is clear that highly interactive Web documents must provide a more sophisticated interface between browsers and mathematical notation. Another problem with encoding mathematics as images is that it requires more bandwidth. Markup describing an equation is typically smaller and more compressible than an image of the equation. Also by using markup-based encoding, more of the rendering process is moved to the client machine.

1.2.4 Requirements for Mathematics Markup

Some display problems associated with including mathematical notation in HTML documents as images could be addressed by improving image handling by browsers. However, even if image handling were improved, the problem of making the information contained in mathematical expressions available to other applications would remain. Therefore, in planning for the future, it is not sufficient merely to upgrade image-based methods. To integrate mathematical material fully into Web documents, a markup-based encoding of mathematical notation and content is required.

In designing any markup language, it is essential to consider carefully the needs of its potential users. In the case of MathML, the needs of potential users cover a broad spectrum, from education to research, and on to commerce:

The education community is a large and important group that must be able to put scientific curriculum materials on the Web. At the same time, educators often have limited resources of time and equipment, and are severely hampered by the difficulty of authoring technical Web documents. Students and teachers need to be able to create mathematical content quickly and easily, using intuitive, easy-to-learn, low-cost tools.

Electronic textbooks are another way of using the Web which will potentially be very important in education. Management consultant Peter Drucker has prophesied the end of big-campus residential higher education and its distribution over the Web [Drucker1997]. Electronic textbooks will need to be interactive, allowing intercommunication between the text and scientific software and graphics.

The academic and commercial research communities generate large volumes of dense scientific material. Increasingly, research publications are being stored in databases, such as the highly successful physics preprint server and archive, www.arXiv.org, which replaces xxx.lanl.gov, at Los Alamos National Laboratory. This is especially true in some areas of physics and mathematics where academic journal prices have been increasing at an unsustainable rate. In addition, databases of information on mathematical research, such as Mathematical Reviews and Zentralblatt für Mathematik, offer millions of records on the Web containing mathematics.

To accommodate the research community, a design for mathematical markup must facilitate the maintenance and operation of large document collections, for which automatic searching and indexing are important. Because of the large collection of legacy documents, for mathematics especially in TeX, the ability to convert between existing formats and any new one is also very important to the research community. Finally, the ability to maintain information for archival purposes is vital to academic research.

Corporate and academic scientists and engineers also use technical documents in their work to collaborate, to record results of experiments and computer simulations, and to verify calculations. For such uses, mathematics on the Web must provide a standard way of sharing information that can be easily read, processed and generated using commonly available, easy-to-use tools.

Another general design requirement is the ability to render mathematical material in other media such as speech or braille, which is extremely important for the visually impaired.

Commercial publishers are also involved with mathematics on the Web at all levels from electronic versions of print books to interactive textbooks and academic journals. Publishers require a method of putting mathematics on the Web that is capable of high-quality output, robust enough for large-scale commercial use, and preferably compatible with their previous, often SGML-based, production systems.

1.2.5 Design Goals of MathML

In order to meet the diverse needs of the scientific community, MathML has been designed with the following ultimate goals in mind.

MathML should:

- Encode mathematical material suitable for teaching and scientific communication at all levels.
- 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 TEX
  - plain text displays, e.g. VT100 emulators
  - 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 template and other mathematics editing techniques.
- Be human legible, and simple for software to generate and process.

No matter how successfully MathML may achieve its goals as a markup language, it is clear that MathML will only be useful if it is implemented well. To this end, 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 equations in HTML 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 documents containing MathML equations should print properly and at high-quality printer resolutions.
- MathML equations in Web pages should be able to react to user gestures, such those as with a mouse, and coordinate communication with other applications through the browser.
- Equation editors and converters should be developed to facilitate the creation of Web pages containing MathML equations.

These goals have begun to be addressed for the near term by using embedded elements such as Java applets, plug-ins and ActiveX controls to render MathML. However, the extent to which these goals are ultimately met depends on the cooperation and support of browser vendors, and other software developers. The W3C Math working group has continued to work with the working groups for the Document Object Model (DOM) and the Extensible Style Language (XSL) to ensure that the needs of the scientific community will be met in the future, and feels that MathML 2.0 shows considerable progress in this area over the situation that obtained at the time of the MathML 1.0 Recommendation (April 1998).

1.3 The Role of MathML on the Web

1.3.1 Layered Design of Mathematical Web Services

The design goals of MathML require a system for encoding mathematical material for the Web which is flexible and extensible, suitable for interaction with external software, and capable of producing high-quality rendering in several media. Any markup language that encodes enough information to do all these tasks well will of necessity involve some complexity.

At the same time, it is important for many groups, such as students, to have simple ways to include mathematics in Web pages by hand. Similarly, other groups, such as the TEX community, would be best served by a system which allowed the direct entry of markup languages like TEX into Web pages. In general, specific user groups are better served by specialized kinds of input and output tailored to their needs. Therefore, the ideal system for communicating mathematics on the Web should provide both specialized services for input and output, and general services for interchange of information and rendering to multiple media.

In practical terms, the observation that mathematics on the Web should provide for both specialized and general needs naturally leads to the idea of a layered architecture. One layer consists of powerful, general software tools exchanging, processing and rendering suitably encoded mathematical data. A second layer consists of specialized software tools, aimed at specific user groups, which are capable of easily generating encoded mathematical data which can then be shared with a particular audience.

MathML is designed to provide the encoding of mathematical information for the bottom, more general layer in a two-layer architecture. It is intended to encode complex notational and semantic structure in an explicit, regular, and easy-to-process way for renderers, searching and indexing software, and other mathematical applications.

As a consequence, raw MathML markup is *not* primarily intended for direct use by authors. While MathML is human-readable, which helps a lot in debugging it, in all but the simplest cases it is too verbose and error-prone for hand generation. Instead, it is anticipated that authors will use equation editors, conversion programs, and other specialized software tools to generate MathML. Alternatively, some renderers may convert other kinds of input directly included in Web pages into MathML on the fly, in response to a cut-and-paste operation, for example.

In some ways, MathML is analogous to other low-level, communication formats such as Adobe's PostScript language. You can create PostScript files in a variety of ways, depending on your needs; experts write and modify them by hand, authors create them with word processors, graphic artists with illustration programs, and so on. Once you have a PostScript file, however, you can share it with a very large audience, since devices which render PostScript, such as printers and screen previewers, are widely available.

Part of the reason for designing MathML as a markup language for a low-level, general, communication layer is to stimulate mathematical Web software development in the layer above. MathML provides a way of coordinating the development of modular authoring tools and rendering software. By making it easier to develop a functional piece of a larger system, MathML can stimulate a 'critical mass' of software development, greatly to the benefit of potential users of mathematics on the Web.

One can envision a similar situation for mathematical data. Authors are free to create MathML documents using the tools best suited to their needs. For example, a student might prefer to use a menu-driven equation editor that can write out MathML to an HTML file. A researcher might use a computer algebra package that automatically encodes the mathematical content of an expression, so that it can be cut from a Web page and evaluated by a colleague. An academic journal publisher might use a program that converts TEX markup to HTML and MathML. Regardless of the method used to create a Web page containing MathML, once it exists, all the advantages of a powerful and general communication layer become available. A variety of MathML software could all be used with the same document to render it in speech or print, to send it to a computer algebra system, or to manage it as part of a large Web document collection. To render high-quality printed mathematics the MathML encoding will often be converted back to standard typesetting and composition languages, including TEX which is widely appreciated for the job it does in this regard. Finally, one may expect that eventually MathML will be integrated into other arenas where mathematical formulas occur, such as spreadsheets, statistical packages and engineering tools.

The W3C Math working group has been working with vendors to ensure that a variety of MathML software will soon be available, including both rendering and authoring tools. A current list of MathML software is maintained on the public Math page at the World Wide Web Consortium .

1.3.2 Relation to Other Web Technology

The original conception of HTML Math was a simple, straightforward extension to HTML that would be natively implemented in browsers. However, very early on, the explosive growth of the Web made it clear that a general extension mechanism was required, and that mathematics was only one of many kinds of structured data which would have to be integrated into the Web using such a mechanism.

Given that MathML must integrate into the Web as an extension, it is extremely important that MathML and MathML software can interact well with the existing Web environment. In particular, MathML has been designed with three kinds of interaction in mind. First, in order to create mathematical Web content, it is important that existing mathematical markup languages can be converted to MathML, and that existing authoring tools can be modified to generate MathML. Second, it must be possible to embed MathML markup seamlessly in HTML markup in such a way that it will be accessible to future browsers, search engines, and all kinds of Web applications which now manipulate HTML. Finally, it must be possible to render MathML embedded in HTML in today's Web browsers in some fashion, even if it is less than ideal.

1.3.2.1 Existing Mathematical Markup Languages

Perhaps the most important influence on mathematical markup languages of the last two decades is the TEX typesetting system developed by Donald Knuth [Knuth1986]. TEX is a de facto standard in the mathematical research community, and it is pervasive in the scientific community at large. TEX sets a standard for quality of visual rendering, and a great deal of effort has gone into ensuring MathML can provide the same visual rendering quality. Moreover, because of the many legacy documents in TEX, and because of the large authoring community versed in TEX, a priority in the design of MathML was the ability to convert TEX mathematics input into MathML format. The feasibility of such conversion has been demonstrated by prototype software.

Extensive work on encoding mathematics has also been done in the SGML community, and SGML-based encoding schemes are widely used by commercial publishers. ISO 12083 is an important markup language which contains a DTD fragment primarily intended for describing the visual presentation of mathematical notation. Because ISO 12083 mathematical notation and its derivatives share many presentational aspects with TEX, and because SGML enforces structure and regularity more than TEX, much of the work in ensuring MathML is compatible with TEX also applies well to ISO 12083.

MathML also pays particular attention to compatibility with other mathematical software, and in particular, with computer algebra systems. Many of the presentation elements of MathML are derived in part from the mechanism of typesetting boxes. The MathML content elements are heavily indebted to the OpenMath project and the work by Stilo Technologies on a mathematical DTD fragment. The OpenMath project has close ties to both the SGML and computer algebra communities, and has laid a foundation for an SGML- and XML-based means of communication between mathematical software packages, amongst other things. The feasibility of both generating and interpreting MathML in computer algebra systems has been demonstrated by prototype software.

1.3.2.2 HTML Extension Mechanisms

As noted above, the success of HTML has led to enormous pressure to incorporate a wide variety of data types and software applications into the Web. Each new format or application potentially places new demands on HTML and on browser vendors. For some time, it has been clear that a general extension mechanism is necessary to accommodate new extensions to HTML. At the very beginning the working group began its work thinking of a plain extension to HTML in the spirit of the first mathematics support suggested for HTML 3.2. But for a good number of reasons, once we got into the details this proved to be not so good an idea. Since work first began on MathML, XML has emerged as the dominant such general extension mechanism.

XML stands for Extensible Markup Language. It is designed as a simplified version of SGML (Standard Generalized Markup Language), the meta-language used to define the grammar and syntax of HTML. One of the goals of XML is to be suitable for use on the Web, and in the context of this discussion it can be viewed as the general mechanism for extending HTML. As its name implies, extensibility is a key feature of XML; authors are free to declare and use new elements and attributes. At the same time, XML grammar and syntax rules carefully enforce regular document structure to facilitate automatic processing and maintenance of large document collections. Mathematically speaking XML is essentially a notation for decorated rooted planar trees, and thus of great generality as an encoding tool.

Since the setting up of the first W3C Math Working Group, XML has garnered broad industry support including that of major browser vendors. The migration of HTML to an XML form has been important to the W3C, and has resulted in the XHTML Recommendation which delivers a new modularized form of HTML. MathML can be viewed as another module which fits very well with the new XHTML. Indeed in Appendix A there is a new DTD for math which is the result of collaboration with the W3C HTML Working Group.

Furthermore, other applications of XML for all kinds of document publishing and processing promise to become increasingly important. Consequently, both on theoretical and pragmatic grounds, it has made a great deal of sense to specify MathML as an XML application.

1.3.2.3 Browser Extension Mechanisms

By now, as opposed to the situation when the MathML 1.0 Recommendation was adopted, the details of a general model for rendering and processing XML extensions to HTML are largely clear. Formatting Properties, developed by the Cascading Style Sheets and Formatting Properties Working Group for CSS and made available through the Document Object Model (DOM), will be applied to MathML elements to obtain stylistic control over the presentation of MathML. Further development of these Formatting Properties falls within the charters of both the CSS&FP and the XSL working groups. For an introduction to this topic see the discussion in Chapter 7. For detailed commentary on how to render MathML with current systems consult the W3C Math WG Home Page.

Until style sheet mechanisms are capable of delivering native browser rendering of MathML, however, it is necessary to extend browser capabilities by using embedded elements to render MathML. It is already possible to instruct a browser to use a particular embedded renderer to process embedded XML markup such as MathML, and to coordinate the resulting output with the surrounding Web page, however the results are not yet entirely as one wishes. See Chapter 7.

For specialized processing, such as connecting to a computer algebra system, the capability of calling out to other programs is likely to remain highly desirable. However, for such an interaction to be really satisfactory, it is necessary to define a document object model rich enough to facilitate complicated interactions between browsers and embedded elements. For this reason, the W3C Math working group has coordinated its efforts closely with the Document Object Model (DOM) working group. The results are described in Chapter 8.

For processing by embedded elements, and for inter-communication between scientific software generally, a style sheet-based layout model is in some ways less than ideal. It can impose an additional implementation burden in a setting where it may offer few advantages, and it imposes implementation requirements for coordination between browsers and embedded renderers that will likely be unavailable in the immediate future.

For these reasons, the MathML specification defines an attribute-based layout model, which has proven very effective for high-quality rendering of complicated mathematical expressions in several independent implementations. MathML presentation attributes utilize W3C Formatting Properties where possible. Also, MathML elements accept `class`, `style` and `id` attributes to facilitate their use with CSS style sheets. However, at present, there are few settings where CSS machinery is currently available to MathML renderers.

The use of CSS style sheet mechanisms has been mentioned above. The mechanisms of XSL have also recently become available for the transformation of XML documents to effect their rendering. Indeed the alternative forms of this present recommendation, including the definitive public HTML version, have been prepared from an underlying XML source using XSL transformation language tools. As further developments in this direction become available to MathML, it is anticipated their use will become the dominant method of stylistic control of MathML presentation meant for use in rendering environments which support those mechanisms.

Chapter 2

MathML Fundamentals

2.1 MathML Overview

This chapter introduces the basic ideas of MathML. The first section describes the overall design of MathML. The second section presents a number of motivating examples, to give the reader something concrete to refer to while reading subsequent chapters of the MathML Specification. The final section describes basic features of the MathML syntax and grammar, which apply to all MathML markup. In particular, Section 2.3 should be read *before* Chapter 3, Chapter 4 and Chapter 5.

A fundamental challenge in defining a markup language for mathematics on the Web is reconciling the need to encode both the presentation of a mathematical notation and the content of the mathematical idea or object which it represents.

The relationship between a mathematical notation and a mathematical idea is subtle and deep. On a formal level, the results of mathematical logic raise unsettling questions about the correspondence between systems of symbolic logic and the phenomena they model. At a more intuitive level, anyone who uses mathematical notation knows the difference that a good choice of notation can make; the symbolic structure of the notation suggests the logical structure. For example, the Leibniz notation for derivatives 'suggests' the chain rule of calculus through the symbolic cancellation of fractions: $\frac{df}{dx}\frac{dx}{dt} = \frac{df}{dt}$.

Mathematicians and teachers intuitively understand this very well; part of their expertise lies in choosing notation that emphasizes key aspects of a problem while hiding or diminishing extraneous aspects. It is commonplace in mathematics and science to write one thing when strictly technically something else is meant, because long experience shows this actually communicates the idea better at some higher level than rigorous detail.

In many other settings, though, mathematical notation is used to encode the full, precise meaning of a mathematical object. Mathematical notation is capable of prodigious rigor, and when used carefully, it can be virtually free of ambiguity. Moreover, it is precisely this lack of ambiguity which makes it possible to describe mathematical objects so that they can be used by software applications such as computer algebra systems and voice renderers. In situations where such inter-application communication is of paramount importance, the nuances of visual presentation generally play a minimal role.

MathML allows authors to encode both the notation which represents a mathematical object and the mathematical structure of the object itself. Moreover, authors can mix both kinds of encoding in order to specify both the presentation and content of a mathematical idea. The remainder of this section gives a basic overview of how MathML can be used in each of these ways.

2.1.1 Taxonomy of MathML Elements

All MathML elements fall into one of three categories: presentation elements, content elements and interface elements. Each of these categories is described in detail in Chapter 3, Chapter 4 and Chapter 7, respectively.

Presentation elements describe mathematical notation's visually oriented two-dimensional structure. Typical examples are the `mrow` element, which is usually employed to indicate a horizontal row of pieces of expressions, and the `msup` element, which is used to mark up a base expression and a superscript to it. As a general rule, each presentation element corresponds to a single kind of '???' such as digits, letters, or other symbol characters.

Although this particular example involves mathematical notation, and hence presentation markup, the same observation about decomposition applies equally well to abstract mathematical objects, and hence to content markup. For example, in the context of content markup our superscript example would typically be denoted by an exponentiation operation that would require two operands: a 'base' and an 'exponent'. This is no coincidence, since as a general rule, mathematical notation's layout closely follows the logical structure of the underlying mathematical objects.

The recursive nature of mathematical objects and notation is strongly reflected in MathML markup. In use, most presentation or content elements contain some number of other MathML elements corresponding to the constituent pieces out of which the original object is recursively built. The original schema is commonly called the *parent* schema, and the constituent pieces are called *child* schemata. More generally, MathML expressions can be regarded as trees, where each node corresponds to a MathML element, the branches under a 'parent' node correspond to its 'children', and the leaves in the tree correspond to atomic notation or content units such as numbers, characters, etc.

Most leaf nodes in a MathML expression tree are either *canonically empty elements* with no bodies, or *token elements*. Canonically empty elements represent symbols directly in MathML, for example, the content element `<plus/>` does this. MathML token elements are the only MathML elements permitted to contain MathML character data. The MathML character data may consist of Unicode characters and MathML `<mchar/>` elements. These `<mchar/>` elements, such as `<mchar name="alpha" />` and `<mchar name="rightarrow" />`, typically denote Unicode characters not in ASCII code and the `name` attribute carries the information as to which special symbol is being represented. The `<mchar/>` construction supersedes the use of special MathML entities such as `α` to encode special symbols specified in MathML 1 for compatibility with general XML mechanisms. A third kind of leaf node permitted in MathML is the `annotation` element, which is used to hold data which is not in MathML format.

The most important presentation token elements are `mi`, `mn` and `mo` for representing identifiers, numbers and operators respectively. Typically a renderer will employ slightly different typesetting styles for each of these kinds of character data: numbers are usually in upright font, identifiers in italics, and operators have extra space around them. In content markup, there are only three tokens, `ci`, `cn` and `csymbol`, for identifiers, numbers and new symbols introduced in the document itself, respectively. In content markup, separate elements are provided for commonly used functions and operators. The `fn` element is provided for user-defined extensions to the base set.

In terms of markup, most MathML elements are denoted by *start* tag and an *end* tag, which enclose the markup for their contents. In the case of tokens, the content is character data, and in most other cases, the content is the markup for child elements. A third category of elements, called canonically empty elements, don't require any contents, and denoted by a single tag of the form `<name/>`. An example of this kind of markup is `<plus/>` in content markup.

Returning to the example of $(a + b)^2$, we can now see how the principles discussed above play out in practice. One form of presentation markup for this example is:

```
<msup>
  <mfenced>
    <mrow>
      <mi>a</mi>
      <mo>+</mo>
      <mi>b</mi>
    </mrow>
  </mfenced>
  <mn>2</mn>
</msup>
```

This example demonstrates a number of presentation elements. The first element, one that is used a great deal is `mrow`. This element is used to denote a row of horizontally aligned material. The material contained between the `<mrow>` and `</mrow>` tags is considered to be an argument to the `mrow` element. Thus the whole expression here is contained in an `mrow` element. As previously noted, almost all mathematical expressions decompose into subexpressions. These subexpressions can can, in turn, also be contained in an `mrow` element. For example, a+b is also contained in an `mrow`.

The `mfenced` element is used to provide fences (braces, brackets, and parentheses) around formula material. It defaults to using parentheses.

Note the use of the `mi` element for displaying the variables a and b and the `mo` element for marking the + operator.

The `msup` element is for expressions involving superscripts and takes two arguments, in order, the base expression (here, $(a+b)$) and the exponent expression (here, 2).

The content markup for the same example is:

```
<apply>
  <power/>
  <apply>
    <plus/>
    <ci>a</ci>
    <ci>b</ci>
  </apply>
  <cn>2</cn>
</apply>
```

Here, the `apply` content element means apply an operation to an expression. In this example, the `power` element (for exponentiation), which requires no body, and the similar `plus` element (for addition) are both *applied*. Observe that both operators take two arguments, the order being particularly significant in the case of the power operator.

Note the use of the `ci` element to denote the variables a and b, and the `cn` element for denoting the number 2.

2.1.2 Presentation Markup

MathML presentation markup consists of about 30 elements which accept over 50 attributes. Most of the elements correspond to *layout schemata*, which contain other presentation elements. Each layout schema corresponds to a two-dimensional notational device, such as a superscript or subscript, fraction or table. In addition, there are the presentation token elements `mi`, `mn` and `mo` introduced above, as well as several other less commonly used token elements. The remaining few presentation elements are empty elements, and are used mostly in connection with alignment.

The layout schemata fall into several classes. One group of elements is concerned with scripts, and contains elements such as `msub`, `munder`, and `mmultiscripts`. Another group focuses on more general layout and includes `mrow`, `mstyle`, and `mfrac`. A third group deals with tables. The `maction` element is in a category by itself, and allows coding of various kinds of actions on notation, such as occur in an expression which toggles between two pieces of notation.

An important feature of many layout schemata is that the order of child schemata is significant. For example, the first child of an `mfrac` element is the numerator and the second child is the denominator. Since the order of child schemata is not enforced at the XML level by the MathML DTD, the information added by ordering is only available to a MathML processor, as opposed to a generic XML processor. When we want to emphasize that a MathML element such as `mfrac` requires children in a specific order, we will refer to them as *arguments*, and think of the `mfrac` element as a notational 'constructor'.

2.1.3 Content Markup

Content markup consists of about 100 elements accepting roughly a dozen attributes. The majority of these elements are empty elements corresponding to a wide variety of operators, relations and named functions. Examples of this sort include `partialdiff`, `leq` and `tan`. Others such as `matrix` and `set` are used to encode various mathematical data types, and a third, important category of content elements such as `apply` are used to apply operations to expressions and also to make new mathematical objects from others.

The `apply` element is perhaps the single most important content element. It is used to apply a function or operation to a collection of arguments. The positions of the child schemata are again significant, with the first child denoting the function to be applied, and the remaining children denoting the arguments of the function in order. Note that the `apply` construct always uses prefix notation, like the programming language LISP. In particular, even binary operations like subtraction are marked up by applying a prefix subtraction operator to two arguments. For example, a - b would be marked up as

```
<apply>
   <minus/>
   <ci>a</ci>
   <ci>b</ci>
</apply>
```

A number of functions and operations require one or more quantifiers to be well-defined. For example, in addition to an integrand, a definite integral must specify the limits of integration and the bound variable. For this reason, there are several *qualifier* schemata such as `bvar` and `lowlimit`. They are used with operators such as `diff` and `int`.

The `declare` construct is especially important for content markup that might be evaluated by a computer algebra system. The `declare` element provides a basic assignment mechanism, where a variable can be declared to be of a certain type, with a certain value.

2.1.4 Mixing Presentation and Content

Different kinds of markup will be found most appropriate for different kinds of tasks. Documents written before the world-wide web became important were most often intended only for visual communication of information, so that legacy data is probably best translated into pure presentation markup, since semantic information about what the author meant can only be guessed at heuristically. By contrast, some mathematical applications and pedagogically-oriented authoring tools will likely choose to be entirely content-based. The majority of applications fall somewhere in between these extremes. For these applications, the most appropriate markup is a mixture of both presentation and content markup.

The rules for mixing presentation and content markup derive from the general principle that mixed content should only be allowed in places where it makes sense. For content markup embedded in presentation markup this basically means that any content fragments should be semantically meaningful, and should not require additional arguments or quantifiers to be fully specified. For presentation markup embedded in content markup, this usually means that presentation markup must be contained in a content token element, so that it will be treated as an indivisible notational unit used as a variable or function name.

Another option is to use a `semantics` element. The `semantics` element is used to bind MathML expressions to various kinds of annotations. One common use for the `semantics` element is to bind a piece of content markup to some presentation markup as a semantic annotation. In this way, an author can specify a non-standard notation to be used when displaying a particular content expression. Another use of the `semantics` element is to bind some other kind of semantic specification, such as an OpenMath expression, to a MathML expression. In this way, the `semantics` element can be used to extend the scope of MathML content markup.

2.2 Some MathML Examples

2.2.1 Presentation Examples

Notation: $x^2 + 4x + 4 = 0$.

Markup:

```
<mrow>
  <mrow>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <mo>+</mo>
    <mrow>
      <mn>4</mn>
      <mo>&InvisibleTimes;</mo>
      <mi>x</mi>
    </mrow>
    <mo>+</mo>
    <mn>4</mn>
  </mrow>
  <mo>=</mo>
  <mn>0</mn>
</mrow>
```

The mfrac and msqrt elements are used for generating fractions and square roots, respectively.

Note the use of nested `mrow` elements to denote terms, for example, the left-hand side of the equation functioning as an operand of '='. Marking terms greatly facilitates spacing for visual rendering, voice rendering, and line breaking. The `InvisibleTimes` MathML character entity is used here to indicate to a renderer that there are special spacing rules between the 4 and the x, and that the 4 and the x should not be broken onto separate lines. In fact, this use of an entity is now explicitly deprecated in favor of the use of `<mchar name="InvisibleTimes" />` but was introduced in MathML 1.0. The new version, which was mentioned above, will be used in the examples below, and is explcitly discussed in Section 4.4.1.

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

Markup:

```
<mrow>
  <mi>x</mi>
  <mo>=</mo>
  <mfrac>
    <mrow>
      <mrow>
        <mo>-</mo>
        <mi>b</mi>
      </mrow>
      <mo><mchar name="PlusMinus"/></mo>
      <msqrt>
        <mrow>
          <msup>
            <mi>b</mi>
            <mn>2</mn>
          </msup>
          <mo>-</mo>
          <mrow>
            <mn>4</mn>
            <mo><mchar name="InvisibleTimes"/></mo>
            <mi>a</mi>
            <mo><mchar name="InvisibleTimes"/></mo>
            <mi>c</mi>
          </mrow>
        </mrow>
      </msqrt>
    </mrow>
    <mrow>
      <mn>2</mn>
      <mo><mchar name="InvisibleTimes"/></mo>
      <mi>a</mi>
    </mrow>
  </mfrac>
</mrow>
```

Notice that the 'plus or minus' sign is given by a special element used for specific symbol names `<mchar name="PlusMinus" />`. Then the same construction is used with `<mchar name="InvisibleTimes" />`, instead of the old form in the previous example. MathML provides a very comprehensive list of character names for mathematical symbols. In addition to the mathematical symbols needed for screen and print rendering, MathML provides symbols to facilitate audio rendering. For audio rendering, it is important to be able to automatically determine whether

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

should be read as 'z times the quantity x plus y' or 'z of x plus y'. The markup elements `<mchar name="InvisibleTimes" />` and `<mchar name="ApplyFunction" />` provide a way for authors to directly encode the distinction for audio renderers. For instance, in the first case `<mchar name="InvisibleTimes" />` should be inserted after the line containing the z. MathML also introduces entities like `<mchar name="dd" />` which represents a 'differential d' which renders with slightly different spacing in print, and can be rendered as 'd' or 'with respect to' in speech. Unless content tags, or some other mechanism, are used to eliminate the ambiguity, authors should always use these entities, in order to make their documents more accessible.

Notation: $A = \begin{bmatrix} x & y \\ z & w \end{bmatrix}$.

Markup:

```
<mrow>
  <mi>A</mi>
  <mo>=</mo>
  <mfenced open="[" close="]">
    <mtable>
      <mtr>
         <mtd><mi>x</mi></mtd>
         <mtd><mi>y</mi></mtd>
      </mtr>
      <mtr>
         <mtd><mi>z</mi></mtd>
         <mtd><mi>w</mi></mtd>
      </mtr>
    </mtable>
  </mfenced>
</mrow>
```

Most elements have a number of attributes that control the details of their screen and print rendering. For example, there are several attributes for the `mfenced` element that controls what delimiters should be used at the beginning and the end of the grouped expression above. The attributes for operator elements given using `<mo>` are set to default values determined by a dictionary. For the suggested MathML operator dictionary, see Appendix D.

2.2.2 Content Examples

Notation: $x^2 + 4x + 4 = 0$.

Markup:

```
<apply>
  <eq/>
  <apply>
    <plus/>
    <apply>
      <power/>
      <ci>x</ci>
      <cn>2</cn>
    </apply>
    <apply>
      <times/>
      <cn>4</cn>
      <ci>x</ci>
    </apply>
    <cn>4</cn>
  </apply>
  <cn>0</cn>
</apply>
```

Note that the `apply` element is used for relations, operators and functions.

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

Markup:

```
<apply>
  <eq/>
  <ci>x</ci>
  <apply>
    <divide/>
    <apply>
      <fn><mo><mchar name='PlusMinus'/></mo></fn>
      <apply>
        <minus/>
        <ci>b</ci>
      </apply>
      <apply>
        <root/>
        <apply>
          <minus/>
          <apply>
            <power/>
            <ci>b</ci>
            <cn>2</cn>
          </apply>
          <apply>
            <times/>
            <cn>4</cn>
            <ci>a</ci>
            <ci>c</ci>
          </apply>
        </apply>
        <cn>2</cn>
      </apply>
    </apply>
    <apply>
      <times/>
      <cn>2</cn>
      <ci>a</ci>
    </apply>
  </apply>
</apply>
```

MathML content markup does not directly contain an element for the ‘plus or minus’ operation. Therefore, we use the `fn` element to declare that we want the presentation markup for this operator to act as a content operator. This is a simple example of how presentation and content markup can be mixed to extend content markup.

Notation: $A = \begin{pmatrix} x & y \\ z & w \end{pmatrix}$.

Markup:

```
<apply>
  <eq/>
  <ci>A</ci>
  <matrix>
    <matrixrow>
      <ci>x</ci>
      <ci>y</ci>
    </matrixrow>
    <matrixrow>
      <ci>z</ci>
      <ci>w</ci>
    </matrixrow>
  </matrix>
</apply>
```

Note that, by default, the rendering of the content element `matrix` includes enclosing parentheses, so we need not directly encode them. This is quite different from the presentation element `mtable` which may or may not refer to a matrix, and hence requires explicit encoding of parentheses if they are desired.

2.2.3 Mixed Markup Examples

Notation: $\int_0^t \frac{\mathrm{d}x}{x}$.

Markup:

```
<semantics>
  <mrow>
    <msubsup>
      <mo><mchar name='int'/></mo>
      <mn>0</mn>
      <mi>t</mi>
    </msubsup>
    <mfrac>
      <mrow>
        <mo><mchar name='dd'/></mo>
        <mi>x</mi>
      </mrow>
      <mi>x</mi>
    </mfrac>
  </mrow>
  <annotation-xml encoding="MathML-Content">
    <apply>
      <int/>
      <bvar><ci>x</ci></bvar>
      <lowlimit><cn>0</cn></lowlimit>
      <uplimit><ci>t</ci></uplimit>
      <apply>
        <divide/>
        <cn>1</cn>
```

```
        <ci>x</ci>
      </apply>
    </apply>
  </annotation-xml>
</semantics>
```

In this example, we use the `semantics` element to provide a MathML content expression to serve as a 'semantic annotation' for a presentation expression. The `semantics` element has as its first child the expression being annotated, and the subsequent children are the annotations. There is no restriction on the kind of annotation that can be attached using the `semantics` element. For example, one might give a TEX encoding, or computer algebra input in an annotation. The type of annotation is specified by the `encoding` attribute and the `annotation` and `annotation-xml` elements.

Another common use of the `semantics` element arises when one wants to use a content coding, and provide a suggestion for its presentation. In such a case, applied to the formula above we would have the markup:

```
<semantics>
  <apply>
    <int/>
    <bvar><ci>x</ci></bvar>
    <lowlimit><cn>0</cn></lowlimit>
    <uplimit><ci>t</ci></uplimit>
    <apply>
      <divide/>
      <cn>1</cn>
      <ci>x</ci>
    </apply>
  </apply>
  <annotation-xml encoding="MathML-Presentation">
    <mrow>
      <msubsup>
        <mo><mchar name='int'/></mo>
        <mn>0</mn>
        <mi>t</mi>
      </msubsup>
      <mfrac>
        <mrow>
          <mo><mchar name='dd'/></mo>
          <mi>x</mi>
        </mrow>
        <mi>x</mi>
      </mfrac>
    </mrow>
  </annotation-xml>
</semantics>
```

This kind of annotation is useful when something other than the default rendering of the content encoding is desired. For example, by default, some renderers might layout the integrand something like '$(1/x)$ dx'. Specifying that the integrand should by preference render as 'dx/x' instead can be accomplished with the use of a MathML Presentation annotation as shown. Be aware, however, that renderers are not required to take into account information contained in annotations, and what use is made of them, if any, will depend on the renderer.

2.3 MathML Syntax and Grammar

2.3.1 MathML Syntax and Grammar

MathML is an application of XML, or Extensible Markup Language [Bray1998], and as such its syntax is governed by the rules of XML syntax, and its grammar is in part specified by a DTD, or Document Type Definition. In other words, the details of using tags, attributes, entity references and so on are defined in the XML language specification, and the details about MathML element and attribute names, which elements can be nested inside each other, and so on are specified in the MathML DTD. This is in Appendix A

The W3C in seeking to increase the flexibility of the use of XML for the Web, and to encourage modularization of applications built with XML, has found that the basic form of a DTD is not ideally suited. Therefore a W3C Working Group was created to develop a specification for XML Schemas [XMLSchemas], which are specification documents that will eventually supersede DTDs. MathML 2.0 is consciously designed so that math may take advantage of the latest in the evolving Web technology. Thus there is to be a Schema for MathML. For further information on a MathML Schema see Appendix A and the MathML Home Page ().

However, MathML also specifies some syntax and grammar rules in addition to the general rules it inherits as an XML application. These rules allow MathML to encode a great deal more information than would ordinarily be possible with pure XML, without introducing many more elements, and using a substantially more complex DTD or schema. A grammar for content markup expressions is given in Appendix B. Of course, one drawback to using MathML specific rules is that they are invisible to generic XML processors and validators.

There are basically two kinds of additional MathML grammar and syntax rules. One kind involves placing additional criteria on attribute values. For example, it is not possible in pure XML to require that an attribute value be a positive integer. The second kind of rule specifies more detailed restrictions on the child elements (for example on ordering) than are given in the DTD or even a schema. For example, it is not possible in XML to specify that the first child be interpreted one way, and the second in another.

The following sections discuss features both of XML syntax and grammar in general, and of MathML in particular. Throughout the remainder of the MathML specification, we will usually take care to distinguish between usage required by XML syntax and the MathML DTD (and schema) and usage required by MathML specific rules. However, we will frequently allude to 'MathML errors' without identifying which part of the specification is being violated.

2.3.2 An XML Syntax Primer

Since MathML is an application of XML, the MathML Specification uses the terminology of XML to describe it. Briefly, XML data is composed of Unicode characters (which include ordinary ASCII characters), 'entity references' (informally called 'entities') such as `<` which usually represent 'extended characters', and 'elements' such as `<mi fontstyle="normal"> x </mi>`.

An element quite often encloses other XML data called its 'content', or 'body', between a 'start tag' (sometimes called a 'begin tag') and an 'end tag', much as in HTML. There are also 'empty elements' such as `<plus/>`, whose start tag ends with `/>` to indicate that the element has no content or end tag. The start tag can contain named parameters called 'attributes', such as `fontstyle="normal"` in the example above. For further details on XML, consult the XML specification [Bray1998].

As XML is case-sensitive, MathML element and attribute names are case-sensitive. For reasons of legibility, the MathML defines them almost all in lowercase.

In formal discussions of XML markup a distinction is maintained between an element, such as an `mrow` element, and the tags `<mrow>` and `</mrow>` marking it. What is between the `<mrow>` start tag and the `</mrow>` end tag is the content or body of the `mrow` element. An 'empty element' such as `none` is defined to have no body and so has a single tag of the form `<none/>`. Usually, the distinction between elements and tags will not be so finely drawn in this specification. For instance, we will sometimes refer to the `<mrow>` and `<none/>` elements, really meaning the elements whose tags these are, in order that references to elements are visually distinguishable from references to attributes. However, the words 'element' and 'tag' themselves will be used strictly in accordance with XML terminology.

2.3.3 Children versus Arguments

Many MathML elements require a specific number of child elements or attach additional meanings to children in certain positions. As noted above, these kinds of requirements are MathML specific, and cannot be given entirely using XML syntax and grammar. When the children of a given MathML element are subject to these kinds of additional conditions, we will often refer to them as *arguments* instead of merely as children in order to emphasize their MathML specific usage. Note that especially in Chapter 3 the term 'argument' is usually used in this technical sense, unless otherwise noted, and therefore refers to a child element.

In the detailed discussions of element syntax given with each element throughout the MathML specification, the number of required arguments and their order is implicitly indicated by giving names for the arguments at various positions. This information is also given for presentation elements in the table of argument requirements in Section 3.1.3, and for content elements in Appendix B.

A few elements have other requirements on the number or type of arguments. These additional requirements are described together with the individual elements.

2.3.4 MathML Attribute Values

According to the XML language specification, attributes given to elements must have one of the forms

`attribute-name = "value"`

or

`attribute-name = 'value'`

where whitespace around the '=' is optional.

Attribute names are generally shown in a `monospaced` font within descriptive text in this specification, just as the `monospaced` font is used for examples.

An attribute's value, which in general in MathML can be a string of arbitrary characters, must be surrounded by a pair of either double quotes (`"`) or single quotes (`'`). The kind of quotes not used to surround the value may be included within it.

MathML uses a more complicated syntax for attribute values than the generic XML syntax required by the MathML DTD. These additional rules are intended for use by MathML applications, and it is a MathML error to violate them, though they cannot be enforced by XML processing. The MathML syntax of each attribute value is specified in the table of attributes provided with the description of each element, using a notation described below. In MathML applications these attribute values should be further processed as follows, unless otherwise specified: whitespace is to be ignored except to separate letter and digit sequences into individual words or numbers; and the numerical Unicode references (listed in Chapter 6) which can be used within token elements to represent characters can be used to represent those characters in attribute values (whenever those characters would be permitted by that attribute value's syntax). Note that the use of entity references for most synbols is now deprecated in MathML 2. Thus the use of a numerical Unicode character reference is better, if the input encoding does not allow you to use it directly.

In particular, the characters `"`, `'`, `&` and `<` can be included in MathML attribute values (when permitted by the attribute value syntax) using the entity references `"`, `'`, `&` and `<`, respectively.

The MathML DTD provided in Appendix A declares most attribute value types as `CDATA` strings. This permits increased interoperability with existing SGML and XML software and allows extension to the lists of predefined values. Similar sorts of considerations apply with schemas.

2.3.4.1 Syntax notations used in the MathML specification

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

Notation	What it matches
number	decimal integer or rational number (a string of digits with one decimal point), optionally starting with '-'
unsigned-number	decimal integer or real number, no sign
integer	decimal integer, optionally starting with '-'
positive-integer	decimal integer, unsigned, not 0
string	arbitrary string (always the entire attribute value)
character	single non-whitespace character, or MathML entity reference; whitespace separation is optional
#rrggbb	RGB color value; the three pairs of hexadecimal digits in the example #5599dd define proportions of red, green and blue on a scale of x00 through a strong sky blue.
h-unit	unit of horizontal length (allowable units are listed below)
v-unit	unit of vertical length (allowable units are listed below)
css-fontfamily	explained in CSS subsection, below
css-color-name	explained in CSS subsection, below
other italicized words	explained in the text for each attribute
form +	one or more instances of 'form'
form *	zero or more instances of 'form'
f1 f2 ... fn	one instance of each form, in sequence, perhaps separated by whitespace
f1 | f2 | ... | fn	any one of the specified forms
[form]	an optional instance of 'form'
(form)	same as form
word in plain text	that word, literally present in the attribute value (unless it is obviously part of an explanatory phrase)
quoted symbol	that symbol, literally present in attribute value (e.g. "+" or '+')

The order of precedence of the syntax notation operators is, from highest to lowest precedence:

- form + or form *
- f1 f2 ... fn (sequence of forms)
- f1 | f2 | ... | fn (alternative forms)

A *string* can contain arbitrary characters which are specifiable within XML `CDATA` attribute values; it must use entity references for certain characters, as described earlier. It can contain XML-format entity or character references for any of the characters listed in Chapter 6. No syntax rule in MathML includes a *string* as only part of an attribute value, only as the entire value.

A *character* is a single non-whitespace Unicode character, or a character entity reference, or an expression using the `mchar` and giving a name character. Examples of this last form are given by the hundreds in Chapter 6.

As a simple example, the permissible values of boolean attributes are specified as `true | false`, meaning that the entire attribute value should be either `true` or `false`.

Adjacent keywords and numbers must be separated by whitespace in the actual attribute values, except for unit identifiers (denoted by `h-unit` or `v-unit` syntax symbols) following numbers. Whitespace is not otherwise required, but is permitted between any of the tokens listed above, except (for compatibility with CSS1) immediately before unit identifiers, between the '-' signs and digits of negative numbers, or between `#` or `rrggbb` and `rgb`

Numerical attribute values for dimensions that should depend upon the current font can be given in font-related units, or in named absolute units (described in a separate subsection below). Horizontal dimensions are conventionally given in `em`'s, and vertical dimensions in `ex`'s, by immediately following a number by one of the unit identifiers `em` or `ex`. For example, the horizontal spacing around an operator such as '+' is conventionally given in `em`s, though other units can be used. Using font-related units is usually preferable to using absolute units, since it allows renderings to grow or shrink in proportion to the current font size.

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).

The symbols h-unit, v-unit, css-fontfamily, and css-color-name are explained in the following subsections.

2.3.4.2 Attributes with units

Some attributes accept horizontal or vertical lengths as numbers followed by a 'unit identifier' (often just called a 'unit'). The syntax symbols h-unit and v-unit refer to a unit for horizontal or vertical length, respectively. The possible units and the lengths they refer to are shown in the table below; they are the same for horizontal and vertical lengths, but the syntax symbols are distinguished in attribute syntaxes as a reminder of the direction each is used in.

The unit identifiers and meanings are taken from CSS1. However, the syntax of numbers followed by unit identifiers in MathML is not identical to the syntax of length values with units in CSS style sheets, since numbers in CSS cannot end with decimal points, and are allowed to start with '+' signs.

The possible horizontal or vertical units in MathML are:

Unit identifier	Unit description
em	em (font-relative unit traditionally used for horizontal lengths)
ex	ex (font-relative unit traditionally used for vertical lengths)
px	pixels, or pixel size of 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 default value

The typesetting units em and ex are defined in Appendix F, and discussed further under 'Additional notes' below.

% is a 'relative unit'; when an attribute value is given as n% (for any numerical value n), the value being specified is the default value for the property being controlled multiplied by n divided by 100. The default value (or the way in which it is obtained, when it is not constant) is listed in the table of attributes for each element, and its meaning is described in the subsequent documentation about that attribute. (The mpadded element has its own syntax for % and does not allow it as a unit identifier.)

For consistency with CSS, length units in MathML are rarely optional. When they are, the unit symbol is enclosed in square brackets in the attribute syntax, following the number to which it applies, e.g. number [h-unit]. The meaning of specifying no unit is given in the documentation for each attribute; in general it is that the number given is a multiplier for the default value of the attribute. (In such cases, specifying the number nnn without a unit is equivalent to specifying the number nnn times 100 followed by %. For example, <mo maxsize="2"> (</mo> is equivalent to <mo maxsize="200%"> (</mo>.)

As a special exception (also consistent with CSS), a numerical value equal to 0 need not be followed by a unit identifier even if the syntax specified here requires one. In such cases, the unit identifier (or lack of one) would not matter, since 0 times any unit is 0.

For most attributes, the typical unit which would be used to describe them in typesetting is chosen as the one used in that attribute's default value in this specification; when a specific default value is not given, the typical unit is usually mentioned in the syntax table or in the documentation for that attribute. The most common units are em or ex. However, any unit can be used, unless otherwise specified for a specific attribute.

Additional notes about units

Note that some attributes, e.g. framespacing on a <mtable>, can contain more than one numerical value, each followed by its own unit.

It is conventional to use the font-relative unit `ex` mainly for vertical lengths, and `em` mainly for horizontal lengths, but this is not required. These units are relative to the font and fontsize which would be used for rendering the element in whose attribute value they are specified, which means they should be interpreted *after* attributes such as `fontfamily` and `fontsize` are processed, if those occur on the same element, since changing the current font or fontsize can change the length of one of these units.

The definition of the length of each unit (but not the MathML syntax for length values) is as specified in CSS1, except that if a font provides specific values for `em` and `ex` which differ from the values defined by CSS1 (the font size and 'x'-height respectively), those values should be used.

2.3.4.3 CSS-compatible attributes

Several MathML attributes, listed below, correspond closely to text rendering properties defined by Cascading Style Sheets, Level 1 (CSS1).

The names and acceptable values of these attributes have been aligned with the CSS1 recommendation where possible. In general, the MathML syntax for each attribute is intended to be a subset of the CSS syntax for the corresponding property. Differences of detail, where they exist, are explained with the documentation about each attribute, in the sections of this specification listed in the table above.

The syntax of certain attributes is partially specified, in the tables of attribute syntax in this specification, using one of the symbols `css-fontfamily` or `css-color-name`, as shown in the following table. These symbols refer to syntaxes from other W3C Recommendations, and are explained in the sections of this specification referred to in the table.

MathML attribute	CSS property	syntax symbol	MathML elements	refer to
fontsize	font-size	-	presentation tokens; `mstyle`	Section 3.2.1
fontweight	font-weight	-	presentation tokens; `mstyle`	Section 3.2.1
fontstyle	font-style	-	presentation tokens; `mstyle`	Section 3.2.1
fontfamily	font-family	css-fontfamily	presentation tokens; `mstyle`	Section 3.2.1
color	color	css-color-name	presentation tokens; `mstyle`	Section 3.3.4
background	background	css-color-name	`mstyle`	Section 3.3.4

See also Section 2.3.5 below for a discussion of the `class`, `style` and `id` attributes for use with style sheets.

Order of processing attributes versus style sheets

CSS or analogous style sheets can specify changes to rendering properties of selected MathML elements; the selection of the elements can happen in various ways). Either the properties listed above, or other MathML rendering attributes or properties supported by a style sheet mechanism, can be affected, in principle, for any element. Since rendering properties can also be changed by attributes on an element or automatically it is necessary to specify the order in which changes from various sources occur. An example of automatic adjustment is what happens for `fontsize`, as explained in the discussion on `scriptlevel` in Section 3.3.4. In the case of 'absolute' changes, i.e. setting a new property value independent of the old value (as opposed to 'relative' changes, such as increments or multiplications by a factor), the absolute change performed last will be the only absolute change which is effective, so the sources of changes which should have the highest priority must be processed last.

In the case of CSS1, the order of processing of changes from various sources which affect one MathML element's rendering properties should be as follows:

(first changes; lowest priority)

- automatic changes to properties or attributes based on the type of the parent element, and this element's position in the parent, as for the changes to `fontsize` in relation to `scriptlevel` mentioned above; such changes will usually be implemented by the parent element itself before it passes a set of rendering properties to this element
- from a style sheet from the reader: styles which are *not* declared 'important'
- explicit attribute settings on this MathML element
- from a style sheet from the author: styles which are *not* declared 'important'
- from a style sheet from the reader: styles which *are* declared 'important'
- from a style sheet from the author: styles which *are* declared 'important'

(last changes; highest priority)

Note that the order of the changes derived from CSS style sheets is specified by CSS itself. The following rationale is related only to the issue of where in this pre-existing order the changes caused by explicit MathML attribute settings should be inserted.

Rationale: MathML rendering attributes are analogous to HTML rendering attributes such as `align`, which the CSS1 section on cascading order specifies should be processed with the same priority. Furthermore, this choice of priority permits readers, by declaring certain CSS styles as 'important', to decide which of their style preferences should override explicit attribute settings in MathML. Since MathML expressions, whether composed of 'presentation' or 'content' elements, are primarily intended to convey meaning, with their 'graphic design' (if any) intended mainly to aid in that purpose but not to be essential in it, it is likely that readers will often want their own style preferences to have priority; the main exception will be when a rendering attribute is intended to alter the meaning conveyed by an expression, which is generally discouraged in the presentation attributes of MathML.

2.3.4.4 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 (unless they are obviously explanatory phrases), 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 under `mstyle`, or in some cases (described individually) 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 rendering environment.

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

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 shown around example attribute values in the text.

Note that, in general, there is no value which can be given explicitly for a MathML attribute which will simulate the effect of not specifying the attribute at all for attributes which are *inherited* or *automatic*. Giving the words 'inherited' or 'automatic' explicitly will not work, and is not generally allowed. Furthermore, even for presentation attributes for which a specific default value is documented here, the `mstyle` element (Section 3.3.4) can be used to change this for the elements it contains. Therefore, the MathML DTD declares most presentation attribute default values as `#IMPLIED`, which prevents XML preprocessors from adding them with any specific default value. This point of view is carried through to the MathML schema

2.3.4.5 Attribute values in the MathML DTD

In an XML DTD, allowed attribute values can be declared as general strings, or they can be constrained in various ways, either by enumerating the possible values, or by declaring them to be certain special data types. The choice of an XML attribute type affects the extent to which validity checks can be performed using a DTD.

The MathML DTD specifies formal XML attribute types for all MathML attributqes, including enumerations of legitimate values in some cases. In general, however, the MathML DTD is relatively permissive, frequently declaring attribute values as strings; this is done to provide for interoperability with SGML parsers while allowing multiple attributes on one MathML element to accept the same values (such as `true` and `false`), and also to allow extension to the lists of predefined values.

At the same time, even though an attribute value may be declared as a string in the DTD, only certain values are legitimate in MathML, as described above and in the rest of this specification. For example, many attributes expect numerical values. In the sections which follow, the allowed attribute values are described for each element. To determine when these constraints are actually enforced in the MathML DTD, consult Appendix A. However, lack of enforcement of a requirement in the DTD does *not* imply that the requirement is not part of the MathML language itself, or that it will not be enforced by a particular MathML renderer. (See Section 7.2.2 for a description of how MathML renderers should respond to MathML errors.)

Furthermore, the MathML DTD is provided for convenience; although it is intended to be fully compatible with the text of the specification, the text should be taken as definitive if there is a contradiction. (Any contradictions which may exist between various chapters of the text should be resolved by favoring Chapter 6 first, then Chapter 3, Chapter 4, then Section 2.3, and then other parts of the text.) For the MathML Schema the situation will be the same: the published Recommendation text takes precedence. Though this is what is intended to happen, there is a practical difficulty. If the system processing the MathML uses a validating parser, whether it be based on a DTD or on a Schema, the process will probably simply stop when it hits something held to be incorrect syntax, whether or not further MathML processing in full harmony with the specification would have processed the piece correctly.

2.3.5 Attributes Shared by all MathML Elements

In order to facilitate compatibility with Cascading Style Sheets, Level 1 (CSS1), all MathML elements accept `class`, `style`, and `id` attributes in addition to the attributes described specifically for each element. MathML renderers not supporting CSS may ignore these attributes. MathML specifies these attribute values as general strings, even if style-sheet mechanisms have more restrictive syntaxes for them. That is, any value for them is valid in MathML.

Renderers supporting CSS (or analogous style sheet mechanisms) may use these attributes to help determine which MathML elements should be subject to which style sheet-induced changes to various rendering properties. The properties that can be affected, and how these changes affect them, are discussed in Section 2.3.4.3 above.

Every MathML element, because of a legacy from MathML 1.0, also accepts the deprecated attribute `other` (Section 7.2.3) which was conceived for passing non-standard attributes without violating the MathML DTD. MathML renderers are only required to process this attribute if they respond to any attributes which are not standard in MathML. However, the use of `other` is strongly deprecated when there are already other ways within MathML of passing specific information.

See also Section 3.2.1 for a list of MathML attributes which can be used on most presentation token elements.

2.3.6 Collapsing Whitespace in Input

MathML ignores whitespace occurring outside token elements. Non-whitespace characters are not allowed there. Whitespace occurring within the content of token elements is ‘trimmed’ from the ends, i.e. all whitespace at the beginning and end of the content is removed. Whitespace internal to content of MathML elements is ‘collapsed’ canonically, i.e. each sequence of 1 or more whitespace characters is replaced with one space character (sometimes called a blank character).

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.

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

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

is equivalent to `<mtext>Theorem 1:</mtext>`.

Authors wishing to encode whitespace 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 ` ` or other ‘whitespace’ non-marking entities as described in Section 6.1.4. For example, compare

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

with

```
<mtext>
 Theorem&NewLine;  1:
</mtext>
```

When the first example is rendered, there is no whitespace before ‘Theorem’, one space between ‘Theorem’ and ‘1:’, and no whitespace after ‘1:’. In the second example, a single space is rendered before ‘Theorem’, a new line is placed after ‘Theorem’, two spaces are rendered before ‘1:’, and there is no whitespace after the ‘1:’.

Note that the `xml:space` attribute does not apply in this situation since XML processors pass whitespace in tokens to a MathML processor; it is the MathML processing rules 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` and `annotation`, an `mspace` element should be used, as opposed to an `mtext` element containing only ‘whitespace’ entities.

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, for example, a single row of identifiers and operators, such as '$x + a / b$', will often be represented not just by one `mrow` element (which renders as a horizontal row of its arguments), but by multiple nested `mrow` elements corresponding to the nested sub-expressions of which one mathematical expression is composed - in this case,

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

Similarly, superscripts are attached not just to the preceding character, but to the full expression constituting their base. This structure allows for better-quality rendering of mathematics, especially when details of the rendering environment such as display widths are not known to the document author; it also greatly eases automatic interpretation of the mathematical structures being represented.

Certain MathML characters are used to name operators or identifiers that in traditional notation render the same as other symbols, such as `ⅆ`, `ⅇ`, or `ⅈ`, or operators that usually render invisibly, such as `⁢`, `⁡`, or `⁣`. These are distinct notational symbols or objects, as evidenced by their distinct spoken renderings and in some cases by their effects on linebreaking and spacing in visual rendering, and as such should be represented by the appropriate specific entity references. For example, the expression represented visually as '$f(x)$' would usually be spoken in English as 'f of x' rather than just '$f\ x$'; this is expressible in MathML by the use of the `⁡` operator after the 'f', which (in this case) can be aurally rendered as 'of'.

The complete list of MathML entities is described in Chapter 6.

3.1.2 Terminology Used In This Chapter

It is strongly recommended that, before reading the present chapter, one read Section 2.3 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 2.3.2, and the attribute value notations and conventions described in Section 2.3.4.

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, etcetera. In general, tokens can have only characters and `mchar` elements as content. The only exceptions are the vertical alignment element `malignmark`, and entity references. (Note, however, that entity references are deprecated in favor of the `mchar` element in MathML 2.0 .) *Layout schemata* build expressions out of parts, and can have only elements as content (except for whitespace, which they ignore). 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 for representing text or whitespace that has more aesthetic than mathematical significance, and for 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, etcetera), 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 3.1.5. The *content elements* are the MathML elements defined in Chapter 4. The content elements are listed in Section 4.4.

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 (listed in Section 5.2.4). The intuition behind the definition of an expression is that it is an element with an unambiguous rendering without some larger, enclosing construct. 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 non-overlapping 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 3.2.6 for a discussion of the proper use of such space-like elements.

3.1.3.1 Inferred `mrows`

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

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.

In the descriptions in this chapter of the above-listed elements' rendering behaviors, their content can be assumed to consist of exactly one expression, which may be an `mrow` element formed from their arguments in this manner. However, their argument counts are shown in the following table as 1*, since they are most naturally understood as acting on a single expression.

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.

Element	Required argument count	Argument roles (when these differ by position)
mrow	0 or more	
mfrac	2	*numerator denominator*
msqrt	1*	
mroot	2	*base index*
mstyle	1*	
merror	1*	
mpadded	1*	
mphantom	1*	
mfenced	0 or more	
menclose	1*	
msub	2	*base subscript*
msup	2	*base superscript*
msubsup	3	*base subscript superscript*
munder	2	*base underscript*
mover	2	*base overscript*
munderover	3	*base underscript overscript*
mmultiscripts	1 or more	*base* (*subscript superscript*)* [<mprescripts/> (*presubscript presuperscript*)*]
mtable	0 or more rows	0 or more mtr or mlabeledtr elements
mlabeledtr	1 or more	a label and 0 or more mtd elements
mtr	0 or more	0 or more mtd elements
mtd	1*	
maction	1 or more	depend on actiontype attribute

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 3.2.6. This definition affects some of the suggested rendering rules for mo elements (Section 3.2.4).

Certain elements, e.g. msup, are able to embellish operators that are their first argument. These elements are listed in Section 3.2.4, which precisely defines an ‘embellished operator’ and explains how this affects the suggested rendering rules for stretchy operators.

Certain elements treat their arguments as the arguments of an ‘inferred mrow’ if they are not given exactly one argument, as explained in Section 3.1.3.

In MathML 1.x, the mtable element could infer mtr elements around its arguments, and the mtr element could infer mtd elements. In MathML 2.0, mtr and mtd elements must be explicit. However, for backward compatibility renderers may wish to continue supporting inferred mtr and mtd elements.

3.1.5 Summary of Presentation Elements

3.1.5.1 Token Elements

mi	identifier
mn	number
mo	operator, fence, or separator
mtext	text
mspace	space
ms	string literal
mchar	referring to non-ASCII characters
mglyph	adding new character glyphs to MathML

3.1.5.2 General Layout Schemata

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

3.1.5.3 Script and Limit Schemata

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

3.1.5.4 Tables and Matrices

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

3.1.5.5 Enlivening Expressions

maction	bind actions to a sub-expression

3.2 Token Elements

Token elements can contain any sequence of zero or more MathML characters. In particular, tokens with empty content are allowed, and should typically render invisibly, with no width except for the normal extra spacing for that kind of token element.

The complete set of MathML characters described in Chapter 6. In general, MathML characters can be either represented directly as Unicode character data, or indirectly via the `&mchar;` element (see Section 3.2.8). MathML characters can also be represented via entity references, although this practice is deprecated in MathML 2.0. New MathML characters, or non-standard glyphs for existing MathML characters, may be represented by means of the `&mglyph;` element.

MathML characters are only allowed to occur as part of the content of a token element. The only exception is whitespace between elements, which is ignored.

Apart from the `&mchar;` and `&mglyph;` elements, the `malignmark` element is the only other element allowed in the content of tokens. See Section 3.5.5 for details.

3.2.1 Attributes common to token elements

Several attributes related to text formatting are provided on all presentation token elements except `mspace`, `mchar` and `mglyph`, and on no other elements except `mstyle`. These are:

Name	values	default
fontsize	number v-unit	inherited
fontweight	normal | bold	inherited
fontstyle	normal | italic	normal (*except on* `<mi>`)
fontfamily	string | css-fontfamily	inherited
color	#rgb | #rrggbb | html-color-name	inherited

(See Section 2.3.4 for terminology and notation used in attribute value descriptions.)

Token elements (other than `mspace`) should be rendered as their content (i.e. in the visual case, as a closely-spaced horizontal row of standard glyphs for the characters in their content) using the attributes listed above, with surrounding spacing modified by rules or attributes specific to each type of token element. Some of the individual attributes are further discussed below.

Recall that all MathML elements, including tokens, accept `class`, `style`, and `id` attributes for compatibility with style sheet mechanisms, as described in Section 2.3.5. In general, the font properties controlled by the attributes listed above are better handled using CSS or XSL style sheets depending on the context.

MathML expressions are often embedded in a textual data format such as HTML, and their renderings are likewise embedded in a rendering of the surrounding text. The renderer of the surrounding text (e.g. a browser) should provide the MathML renderer with information about the rendering environment, including attributes of the surrounding text such as its font size, so that the MathML can be rendered in a compatible 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. (Note that it is also important for the rendering environment to provide the renderer with additional information, such as the baseline position of surrounding text, which is not specified by any MathML attributes.)

The exception to the general pattern of inheritance is the `fontstyle` attribute, whose default value is `normal` (non-slanted) for most tokens, but for `mi` depends on the content in a way described in the section about `mi`, Section 3.2.2. Note that `fontstyle` is not inherited in MathML, even though the corresponding CSS1 property 'font-style' is inherited in CSS.

The `fontsize` attribute specifies the desired font size. `v-unit` represents a unit of vertical length (see Section 2.3.4.3). The most common unit for specifying font sizes in typesetting is `pt` (points).

If the requested size 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.

Many MathML elements automatically change `fontsize` in some of their children; see the discussion of `scriptlevel` in the section on `mstyle`, Section 3.3.4.

The value of the `fontfamily` attribute should be the name of a font that may be available to a MathML renderer, or information that permits the renderer to select a font in some manner; acceptable values and their meanings are dependent on the specific renderer and rendering environment in use, and are not specified by MathML (but see the note about `css-fontfamily` below). (Note that the renderer's mechanism for finding fonts by name may be case-sensitive.)

If the value of `fontfamily` is not recognized by a particular MathML renderer, this should never be interpreted as a MathML error; rather, the renderer should either use a font that it considers to be a suitable substitute for the requested font, or ignore the attribute and act as if no value had been given.

Note that any use of the `fontfamily` attribute is unlikely to be portable across all MathML renderers. In particular, it should never be used to try to achieve the effect of a reference to an non-ASCII MathML character (for example, by using a reference to a character in some symbol font that maps ordinary characters to glyphs for non-ASCII characters). As a corollary to this principle, MathML renderers should attempt to always produce intelligible renderings for the MathML characters listed in Chapter 6, even when these characters are not available in the font family indicated. Such a rendering is always possible - as a last resort, a character can be rendered to appear as an XML-style entity reference using one of the entity names given for the same character in Chapter 6.

The symbol `css-fontfamily` refers to a legal value for the `font-family` property in CSS1, which is a comma-separated list of alternative font family names or generic font types in order of preference, as documented in more detail in CSS1. MathML renderers are encouraged to make use of the CSS syntax for specifying fonts when this is practical in their rendering environment, even if they do not otherwise support CSS. (See also the subsection CSS-compatible attributes within Section 2.3.4.3.

The syntax and meaning of the `color` attribute are as described for the same attribute of `<mstyle>` (Section 3.3.4).

3.2.2 Identifier (`mi`)

3.2.2.1 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.

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 3.2.5.4.

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 like 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.

3.2.2.2 Attributes

`mi` elements accept the attributes listed in Section 3.2.1, but in one case with a different default value:

Name	values	default
fontstyle	normal | italic	(depends on content; described below)

A typical graphical renderer would render an `mi` element as the characters in its content, with no extra spacing around the characters (except spacing associated with neighboring elements). The default `fontstyle` would (typically) be `normal` (non-slanted) unless the content is a single character, in which case it would be `italic`. Note that this rule for `fontstyle` is specific to `mi` elements; the default value for the `fontstyle` attribute of other MathML token elements is `normal`.

3.2.2.3 Examples

```
<mi> x </mi>
<mi> D </mi>
<mi> sin </mi>
<mi></mi>
```

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

Identifiers include function names such as 'sin'. Expressions such as 'sin *x*' should be written using the `⁡` operator (which also has the short name `⁡`) as shown below; see also the discussion of invisible operators in Section 3.2.4.

```
<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> ... </mi>
  <mo> + </mo>
  <mi> n </mi>
</mrow>
```

When an `mi` is used in such exceptional situations, explicitly setting the `fontstyle` 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>
```

Use of special entity references for such constants can simplify the interpretation of MathML presentation elements. See Chapter 6 for a complete list of character entity references in MathML.

3.2.3 Number (mn)

3.2.3.1 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.

The concept of a mathematical 'number' depends on the context, and is not well-defined in the abstract. 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 negative numbers, 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 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.

3.2.3.2 Attributes

`mn` elements accept the attributes listed in Section 3.2.1.

A typical graphical renderer would render an `mn` element as the characters of its content, with no extra spacing around them (except spacing from neighboring elements such as `mo`). Unlike `mi`, `mn` elements are (typically) rendered in an unslanted font by default, regardless of their content.

3.2.3.3 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>
```

3.2.3.4 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>
```

3.2.4 Operator, Fence, Separator or Accent (`mo`)

3.2.4.1 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.

The term 'operator' as used in the present chapter means any symbol or notation that should render as an operator, and that is therefore representable by an `mo` element. That is, the term 'operator' includes any ordinary operator, fence, separator, or accent unless otherwise specified or clear from the context.

All such symbols are represented in MathML with `mo` elements since they 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 3.2.5. See also Chapter 4 for definitions of MathML content elements that are guaranteed to have the semantics of specific mathematical operators.

3.2.4.2 Attributes

`mo` elements accept the attributes listed in Section 3.2.1, and the additional attributes listed here. Most attributes get their default values from the Section 3.2.4.7, as described later in this section. When a dictionary entry is not found for a given `mo` element, the default value shown here in parentheses is used.

Name	values	default
form	prefix | infix | postfix	set by position of operator in an `mrow` (rule given below); used with `mo` content to index operator dictionar
fence	true | false	set by dictionary (false)
separator	true | false	set by dictionary (false)
lspace	number h-unit | namedspace	set by dictionary (thickmathspace)
rspace	number h-unit | namedspace	set by dictionary (thickmathspace)
stretchy	true | false	set by dictionary (false)
symmetric	true | false	set by dictionary (true)
maxsize	number [v-unit | h-unit] | namedspace | infinity	set by dictionary (infinity)
minsize	number [v-unit | h-unit] | namedspace	set by dictionary (1)
largeop	true | false	set by dictionary (false)
movablelimits	true | false	set by dictionary (false)
accent	true | false	set by dictionary (false)

`h-unit` represents a unit of horizontal length, and `v-unit` represents a unit of vertical length (see Section 2.3.4.2). `namedspace` is one of `veryverythinmathspace`, `verythinmathspace`, `thinmathspace`, `mediummathspace`, `thickmathspace`, `verythickmathspace`, or `veryverythickmathspace`. These values are settable by the `mstyle` element which is discussed in Section 3.3.4. The default values of `veryverythinmathspace`... `veryverythickmathspace` are 1/18em...7/18em, respectively.

If no unit is given with `maxsize` or `minsize`, the number is a multiplier of the normal size of the operator in the direction (or directions) in which it stretches. These attributes are further explained below.

Typical graphical renderers show all `mo` elements as the characters of their content, with additional spacing around the element determined from the attributes listed above. Detailed rules for determining operator spacing in visual renderings are described in a subsection below. As always, MathML does not require a specific rendering, and these rules are provided as suggestions for the convenience of implementors.

Renderers without access to complete fonts for the MathML character set may choose not to render an `mo` element as 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> &le </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 <=.)

3.2.4.3 Examples with ordinary operators

```
<mo> + </mo>
<mo> &lt; </mo>
<mo> &le; </mo>
<mo> &lt;= </mo>
<mo> ++ </mo>
<mo> &sum; </mo>
<mo> .NOT. </mo>
<mo> and </mo>
<mo> &InvisibleTimes; </mo>
```

3.2.4.4 Examples with fences and separators

Note that the `mo` elements in these examples don't need explicit `fence` or `separator` attributes, since these can be found using the operator dictionary as described below. Some of these examples could also be encoded using the `mfenced` element described in Section 3.3.8.

$(a+b)$

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

[0,1)

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

$f(x,y)$

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

3.2.4.5 Invisible operators

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

Full name	Short name	Examples of use
`⁢`	`⁢`	xy
`⁡`	`⁡`	$f(x)$ sin x
`⁣`	`⁣`	m_12

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

```
<mrow>
  <mi> x </mi>
  <mo> &InvisibleTimes; </mo>
  <mi> y </mi>
</mrow>
<mrow>
  <mi> f </mi>
  <mo> &ApplyFunction; </mo>
  <mrow>
    <mo> ( </mo>
    <mi> x </mi>
    <mo> ) </mo>
  </mrow>
</mrow>
<mrow>
  <mi> sin </mi>
  <mo> &ApplyFunction; </mo>
  <mi> x </mi>
</mrow>
<msub>
  <mi> m </mi>
  <mrow>
    <mn> 1 </mn>
    <mo> &InvisibleComma; </mo>
    <mn> 2 </mn>
  </mrow>
</msub>
```

The reasons for using specific `mo` elements for invisible operators include:

- such operators should often have specific effects on visual rendering (particularly spacing and linebreaking rules) that are not the same as either the lack of any operator, or spacing represented by `mspace` or `mtext` elements;
- these operators should often have specific audio renderings different than that of the lack of any operator;
- automatic semantic interpretation of MathML presentation elements is made easier by the explicit specification of such operators.

For example, an audio renderer might render $f(x)$ (represented as in the above examples) by speaking ‘f of x’, but use the word ‘times’ in its rendering of xy. Although its rendering must still be different depending on the structure of neighboring elements (sometimes leaving out ‘of’ or ‘times’ entirely), its task is made much easier by the use of a different `mo` element for each invisible operator.

3.2.4.6 Names for other special operators

MathML also includes `ⅆ` for use in an `mo` element representing the differential operator symbol usually denoted by ‘d’. The reasons for explicitly using this special entity are similar to those for using the special entities for invisible operators described in the preceding section.

3.2.4.7 Detailed rendering rules for mo elements

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

Note that, like all rendering rules in MathML, these rules are suggestions rather than requirements. Furthermore, no attempt is made to specify the rendering completely; rather, enough information is given to make the intended effect of the various rendering attributes as clear as possible.

The operator dictionary

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

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

Default value of the form attribute

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

- If the operator is the first argument in an mrow of length (i.e. number of arguments) greater than one (ignoring all space-like arguments (see Section 3.2.6) in the determination of both the length and the first argument), the prefix form is used;
- if it is the last argument in an mrow of length greater than one (ignoring all space-like arguments), the postfix form is used;
- in all other cases, including when the operator is not part of an mrow, the infix form is used.

Note that these rules make reference to the mrow in which the mo element lies. In some situations, this mrow might be an inferred mrow implicitly present around the arguments of an element such as msqrt or mtd.

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

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

Exception for embellished operators

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

For example, the '$+_4$' in $a+_4b$ should be considered an infix operator as a whole, due to its position in the middle of an mrow, but its rendering attributes should be taken from the mo element representing the '+', or when those are not specified explicitly, from the operator dictionary entry for `<mo form="infix"> + </mo>`. The precise definition of an 'embellished operator' is:

- an `mo` element;
- or one of the elements `msub`, `msup`, `msubsup`, `munder`, `mover`, `munderover`, `mmultiscripts`, `mfrac`, or `semantics` (Section 4.2.6), whose first argument exists and is an embellished operator;
- or one of the elements `mstyle`, `mphantom`, or `mpadded`, such that an `mrow` containing the same arguments would be an embellished operator;
- or an `maction` element whose selected sub-expression exists and is an embellished operator;
- or an `mrow` whose arguments consist (in any order) of one embellished operator and zero or more space-like elements.

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

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

Rationale for definition of embellished operators

The following notes are included as a rationale for certain aspects of the above definitions, but should not be important for most users of MathML.

An `mfrac` is included as an ‘embellisher’ because of the common notation for a differential operator:

```
<mfrac>
  <mo> &DifferentialD; </mo>
  <mrow>
    <mo> &DifferentialD; </mo>
    <mi> x </mi>
  </mrow>
</mfrac>
```

Since the definition of embellished operator affects the use of the attributes related to stretching, it is important that it includes embellished fences as well as ordinary operators; thus it applies to any `mo` element.

Note that an `mrow` containing a single argument is an embellished operator if and only if its argument is an embellished operator. This is because an `mrow` with a single argument must be equivalent in all respects to that argument alone (as discussed in Section 3.3.1). This means that an `mo` element that is the sole argument of an `mrow` will determine its default `form` attribute based on that `mrow`’s position in a surrounding, perhaps inferred, `mrow` (if there is one), rather than based on its own position in the `mrow` in which it is the sole argument.

Note that the above definition defines every `mo` element to be ‘embellished’ - that is, ‘embellished operator’ can be considered (and implemented in renderers) as a special class of MathML expressions, of which `mo` is a specific case.

Spacing around an operator

The amount of space added around an operator (or embellished operator), when it occurs in an `mrow`, can be directly specified by the `lspace` and `rspace` attributes. These values are in ems if no units are given. By convention, operators that tend to bind tightly to their arguments have smaller values for spacing than operators that tend to bind less tightly. This convention should be followed in the operator dictionary included with a MathML renderer. In TEX, these values can only be one of three values; typically they are 3/18em, 4/18em, and 5/18em. MathML does not impose this limit.

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

Non-graphical renderers should treat spacing attributes, and other rendering attributes described here, in analogous ways for their rendering medium.

3.2.4.8 Stretching of operators, fences and accents

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

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

The `minsize` and `maxsize` attributes limit the amount of stretching (in either direction). These two attributes are given as multipliers of the operator's normal size in the direction or directions of stretching, or as absolute sizes using units. For example, if a character has `maxsize`="3", then it can grow to be no more than three times its normal (unstretched) size.

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

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

```
<mrow>
  <munder>
    <mo> ( </mo>
    <mo> &UnderBar; </mo>
  </munder>
  <mfrac>
    <mi> a </mi>
    <mi> b </mi>
  </mfrac>
  <mover>
    <mo> ) </mo>
    <mo> &OverBar; </mo>
  </mover>
</mrow>
```

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

Example of stretchy attributes

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

```
<mrow>
  <mo maxsize="1"> ( </mo>
  <mfrac>
    <mi> a </mi> <mi> b </mi>
  </mfrac>
  <mo maxsize="1"> ) </mo>
</mrow>
```

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

Note that each parenthesis is sized independently; if only one of them had `maxsize`="1", they would render with different sizes.

Vertical Stretching Rules

- If a stretchy operator is a direct sub-expression of an `mrow` element, or is the sole direct sub-expression of an `mtd` element in some row of a table, then it should stretch to cover the height and depth (above and below the `axis`) of the non-stretchy direct sub-expressions in the `mrow` element or table row, unless stretching is constrained by `minsize` or `maxsize` attributes.
- In the case of an embellished stretchy operator, the preceding rule applies to the stretchy operator at its core.
- If `symmetric=true`, then the maximum of the height and depth is used to determine the size, before application of the `minsize` or `maxsize` attributes.
- The preceding rules also apply in situations where the `mrow` element is inferred.

Most common opening and closing fences are defined in the operator dictionary to stretch by default; and they stretch vertically. Also, operators such as `∑`, `∫`, /, and vertical arrows stretch vertically by default.

In the case of a stretchy operator in a table cell (i.e. within an `mtd` element), the above rules assume each cell of the table row containing the stretchy operator covers exactly one row. (Equivalently, the value of the `rowspan` attribute is assumed to be 1 for all the table cells in the table row, including the cell containing the operator.) When this is not the case, the operator should only be stretched vertically to cover those table cells that are entirely within the set of table rows that the operator's cell covers. Table cells that extend into rows not covered by the stretchy operator's table cell should be ignored. See Section 3.5.4.2 for details about the the `rowspan` attribute.

Horizontal Stretching Rules

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

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

In the case of a stretchy operator in a table cell (i.e. within an `mtd` element), the above rules assume each cell of the table column containing the stretchy operator covers exactly one column. (Equivalently, the value of the `columnspan` attribute is assumed to be 1 for all the table cells in the table row, including the cell containing the operator.) When this is not the case, the operator should only be stretched horizontally to cover those table cells that are entirely within the set of table columns that the operator's cell covers. Table cells that extend into columns not covered by the stretchy operator's table cell should be ignored. See Section 3.5.4.2 for details about the the `rowspan` attribute.

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

```
<mrow>
  <mi> x </mi>
  <munder>
    <mo> &RightArrow; </mo>
    <mtext> maps to </mtext>
  </munder>
  <mi> y </mi>
</mrow>
```

This displays as $x \underset{\text{maps to}}{\longrightarrow} y$.

Rules Common to both Vertical and Horizontal Stretching

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

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

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

3.2.4.9 Other attributes of `mo`

The `largeop` attribute specifies whether the operator should be drawn larger than normal if `displaystyle=true` in the current rendering environment. This roughly corresponds to TEX's `\displaystyle` style setting. MathML uses two attributes, `displaystyle` and `scriptlevel`, to control orthogonal presentation features that TEX encodes into one 'style' attribute with values `\displaystyle`, `\textstyle`, `\scriptstyle`, and `\scriptscriptstyle`. These attributes are discussed further in Section 3.3.4 describing the `mstyle` element. Note that these attributes can be specified directly on an `mstyle` element's begin tag, but not on most other elements. Examples of large operators include `∫` and `∏`.

The `movablelimits` attribute specifies whether underscripts and overscripts attached to this `mo` element should be drawn as subscripts and superscripts when `displaystyle=false`. `movablelimits=false` means that underscripts and overscripts should never be drawn as subscripts and superscripts. In general, `displaystyle` is `true` for displayed mathematics and `false` for inline mathematics. Also, `displaystyle` is `false` by default within tables, scripts and fractions, and a few other exceptional situations detailed in Section 3.3.4. Thus, operators with `movablelimits=true` will display with limits (i.e. underscripts and overscripts) in displayed mathematics, and with subscripts and superscripts in inline mathematics, tables, scripts and so on. Examples of operators that typically have `movablelimits=true` are `sum`, `prod`, and `lim`.

The `accent` attribute determines whether this operator should be treated by default as an accent (diacritical mark) when used as an underscript or overscript; see `munder`, `mover`, and `munderover` (Section 3.4.4, Section 3.4.5 and Section 3.4.6).

The `separator` attribute may affect automatic linebreaking in renderers that position ordinary infix operators at the beginnings of broken lines rather than at the ends (that is, which avoid linebreaking just after such operators), since linebreaking should be avoided just before separators, but is acceptable just after them.

The `fence` attribute has no effect in the suggested visual rendering rules given here; it is not needed for properly rendering traditional notation using these rules. It is provided so that specific MathML renderers, especially non-visual renderers, have the option of using this information.

3.2.5 Text (`mtext`)

3.2.5.1 Description

An `mtext` element is used to represent arbitrary text that should be rendered as itself. In general, the `mtext` element is intended to denote commentary text that is not central to the mathematical meaning or notational structure of the expression it is contained in.

Note that some text with a clearly defined notational role might be more appropriately marked up using `mi` or `mo`; this is discussed further below.

An `mtext` element can be used to contain 'renderable whitespace', i.e. invisible characters that are intended to alter the positioning of surrounding elements. In non-graphical media, such characters are intended to have an analogous effect, such as introducing positive or negative time delays or affecting rhythm in an audio renderer. This is not related to any whitespace in the source MathML consisting of blanks, newlines, tabs, or carriage returns; whitespace present directly in the source is trimmed and collapsed, as described in Section 2.3.6. Whitespace that is intended to be rendered as part of an element's content must be represented by entity references (unless it consists only of single blanks between non-whitespace characters).

Renderable whitespace can have a positive or negative width, as in ` ` and ``, or zero width, as in ``. The complete list of such characters is given in Chapter 6. Note that there is no formal distinction in MathML between renderable whitespace characters and any other class of characters, in `mtext` or in any other element.

Renderable whitespace can also include characters that affect alignment or linebreaking. Some of these characters are:

Entity name	Purpose (rough description)
NewLine	start a new line and do not indent
IndentingNewLine	start a new line and do indent
NoBreak	do not allow a linebreak here
GoodBreak	if a linebreak is needed on the line, here is a good spot
BadBreak	if a linebreak is needed on the line, try to avoid breaking here

For the complete list of MathML entities, consult Chapter 6.

3.2.5.2 Attributes

`mtext` elements accept the attributes listed in Section 3.2.1.

See also the warnings about the legal grouping of ‘space-like elements’ in Section 3.2.6, and about the use of such elements for ‘tweaking’ or conveying meaning in Section 3.3.6.

3.2.5.3 Examples

```
<mtext> Theorem 1: </mtext>
<mtext>   </mtext>
<mtext>    </mtext>
<mtext> /* a comment */ </mtext>
```

3.2.5.4 Mixing text and mathematics

In some cases, text embedded in mathematics could be more appropriately represented using `mo` or `mi` elements. For example, the expression ‘there exists $\delta > 0$ such that $f(x)$ <1’ is equivalent to $\exists\delta > 0 \ni f(x) < 1$ and could be represented as:

```
<mrow>
  <mo> there exists </mo>
  <mrow>
    <mrow>
      <mi> &delta; </mi>
      <mo> &gt; </mo>
      <mn> 0 </mn>
    </mrow>
    <mo> such that </mo>
    <mrow>
      <mrow>
        <mi> f </mi>
        <mo> &ApplyFunction; </mo>
        <mrow>
          <mo> ( </mo>
          <mi> x </mi>
          <mo> ) </mo>
        </mrow>
      </mrow>
      <mo> &lt; </mo>
      <mn> 1 </mn>
    </mrow>
  </mrow>
</mrow>
```

An example involving an `mi` element is: $x+x^2+\cdots+x^n$. In this example, ellipsis should be represented using an `mi` element, since it takes the place of a term in the sum (see Section 3.2.2, `mi`).

On the other hand, expository text within MathML is best represented with an `mtext` element. An example of this is:

Theorem 1: if $x > 1$, then $x^2 > x$.

However, when MathML is embedded in HTML, or another document markup language, the example is probably best rendered with only the two inequalities represented as MathML at all, letting the text be part of the surrounding HTML.

Another factor to consider in deciding how to mark up text is the effect on rendering. Text enclosed in an `mo` element is unlikely to be found in a renderer's operator dictionary, so it will be rendered with the format and spacing appropriate for an 'unrecognized operator', which may or may not be better than the format and spacing for 'text' obtained by using an `mtext` element. An ellipsis entity in an `mi` element is apt to be spaced more appropriately for taking the place of a term within a series than if it appeared in an `mtext` element.

3.2.6 Space (`mspace`)

3.2.6.1 Description

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

3.2.6.2 Attributes

Name	values	default
width	number h-unit | namedspace	0em
height	number v-unit	0ex
depth	number v-unit	0ex
linebreak	auto | NewLine | IndentingNewLine | NoBreak | GoodBreak | BadBreak	auto

`h-unit` and `v-unit` represent units of horizontal or vertical length, respectively (see Section 2.3.4.2).

The `linebreak` attribute is used to give a linebreaking hint to a visual renderer. The default value is `auto`, which indicates that a renderer should use whatever default linebreaking algorithm it would normally use. The meaning of the other possible values for the `linebreak` attribute are described above in the discussion on renderable whitespace in the `mtext` element. See Section 3.2.5 for details.

In the case when both dimensional attributes and a linebreaking attribute are set, the linebreaking attribute is ignored.

Note the warning about the legal grouping of 'space-like elements' given below, and the warning about the use of such elements for 'tweaking' or conveying meaning in Section 3.3.6. See also the other elements that can render as whitespace, namely `mtext`, `mphantom`, and `maligngroup`.

3.2.6.3 Definition of space-like elements

A number of MathML presentation elements are 'space-like' in the sense that they typically render as whitespace, and do not affect the mathematical meaning of the expressions in which they appear. As a consequence, these elements often function in somewhat exceptional ways in other MathML expressions. For example, space-like elements are handled specially in the suggested rendering rules for `mo` given in Section 3.2.4. The following MathML elements are defined to be 'space-like':

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

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

3.2.6.4 Legal grouping of space-like elements

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

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

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

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

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

See also the warning about 'tweaking' in Section 3.3.6.

3.2.7 String Literal (ms)

3.2.7.1 Description

The ms element is used to represent ‘string literals’ in expressions meant to be interpreted by computer algebra systems or other systems containing ‘programming languages’. By default, string literals are displayed surrounded by double quotes. As explained in Section 3.2.5, ordinary text embedded in a mathematical expression should be marked up with mtext, or in some cases mo or mi, but never with ms.

Note that the string literals encoded by ms are ‘Unicode strings’ rather than ‘ASCII strings’. In practice, non-ASCII characters will typically be represented by mchar elements. For example, `<ms><mchar name="amp"/></ms>` represents a string literal containing a single character, &, and `<ms><mchar name="amp"/>amp;</ms>` represents a string literal containing 5 characters, the first one of which is &.

Like all token elements, ms *does* trim and collapse whitespace in its content according to the rules of Section 2.3.6, so whitespace intended to remain in the content should be encoded as described in that section.

3.2.7.2 Attributes

ms elements accept the attributes listed in Section 3.2.1, and additionally:

Name	values	default
lquote	string	"
rquote	string	"

In visual renderers, the content of an ms element is typically rendered with no extra spacing added around the string, and a quote character at the beginning and the end of the string. By default, the left and right quote characters are both the standard double quote character ". However, these characters can be changed with the lquote and rquote attributes respectively.

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

3.2.8 Referring to non-ASCII characters (mchar)

3.2.8.1 Description

The mchar element is used to reference characters. This provides an alternative to using entity references. Character entities are deprecated for MathML 2.0 because they are not a part of the current proposal for schemas, and documents containing entities are not well-formed XML in the absence of the MathML DTD.

Numeric character references (e.g. Ӓ) are not deprecated because they do not have the problems listed above.

mchar is valid content in any MathML token element listed in Section 3.1.5 (mi, etc.) or Section 4.2.2 (ci, etc.) unless otherwise restricted by an attribute (e.g. base="2" to <cn>).

3.2.8.2 Attributes of mchar

Name	values	default
name	string	required

The name attribute must be one of the names specified in Chapter 6. It is an error to use a name that is not in that list.

3.2.8.3 Examples

In MathML 1.x expressions involving entity references such as <mi> α1 </mi> were common. In MathML 2.0, the equivalent construction using mchar is preferred:

```
<mi> <mchar name='alpha'/>1 </mi>
```

3.2.9 Adding new character glyphs to MathML (mglyph)

3.2.9.1 Description

Unicode defines a large number of characters used in mathematics, and in most all 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 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. In the mean time, a mechanism is necessary for accessing glyphs from non-standard fonts representing these characters.

The `mglyph` element is the means by which users can directly access glyphs for characters that are not defined by Unicode. Similarly, the `mglyph` element can also be used to select glyph variants for existing Unicode characters, as might be desirable when a glyph variant has begun to differentiate itself as a new character by taking on a distinguished mathematical meaning.

The `mglyph` element names a specific character glyph, and is valid inside any MathML leaf content listed in Section 3.1.5 (`mi`, etc.) or Section 4.2.2 (`ci`, etc.) unless otherwise restricted by an attribute (e.g. `base`=2 to `<cn>`). In order for a visually-oriented renderer to render the character, the renderer must be told what font to use and what index within that font to use.

3.2.9.2 Attributes

Name	values	default
alt	string	required
fontfamily	string | css-fontfamily	required
index	integer	required

The `alt` attribute provides an alternate name for the glyph. If the specified font can't be found, 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. The `fontfamily` and `index` uniquely identify the `mglyph`; two `mglyphs` with the same values for `fontfamily` and `index` should be considered identical by applications that must determine whether two characters/glyphs are identical. The `alt` attribute should not be part of the identity test.

The `fontfamily` and `index` attributes name a font and position within that font. All font properties apart from `fontfamily` are inherited. Variants of the font (e.g., bold) that may be inherited may be ignored if the variant of the font is not present.

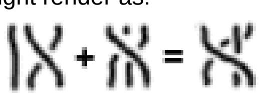
Authors should be aware that rendering requires the fonts referenced by `mglyph`, which the MathML renderer may not have access to or may be not be supported by the system on which the renderer runs. For these reasons, authors are encouraged to use `mglyph` only when absolutely necessary, and not for stylistic purposes.

3.2.9.3 Example

The following example illustrates how a researcher might use the `mglyph` construct with an experimental font to work with braid group notation.

```
<mrow>
  <mi><mglyph fontfamily="my-braid-font" index="2" alt="23braid"/></mi>
  <mo>+</mo>
  <mi><mglyph fontfamily="my-braid-font" index="5" alt="132braid"/></mi>
  <mo>=</mo>
  <mi><mglyph fontfamily="my-braid-font" index="3" alt="13braid"/></mi>
</mrow>
```

This might render as:



3.3 General Layout Schemata

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

3.3.1 Horizontally Group Sub-Expressions (`mrow`)

3.3.1.1 Description

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

Several elements automatically treat their arguments as if they were contained in an `mrow` element. See the discussion of inferred `mrow`s in Section 3.1.3. See also `mfenced` (Section 3.3.8), which can effectively form an `mrow` containing its arguments separated by commas.

3.3.1.2 Attributes

None (except the attributes allowed for all MathML elements, listed in Section 2.3.5).

`mrow` elements are typically rendered visually as a horizontal row of their arguments, left to right in the order in which the arguments occur, or audibly as a sequence of renderings of the arguments. The description in Section 3.2.4 of suggested rendering rules for `mo` elements assumes that all horizontal spacing between operators and their operands is added by the rendering of `mo` elements (or, more generally, embellished operators), not by the rendering of the `mrow`s they are contained in.

MathML is designed to allow renderers to automatically *linebreak* expressions (that is, to break excessively long expressions into several lines), without requiring authors to specify explicitly how this should be done. This is because linebreaking positions can’t be chosen well without knowing the width of the display device and the current font size, which for many uses of MathML will not be known except by the renderer at the time of each rendering.

Determining good positions for linebreaks is complex, and rules for this are not described here; whether and how it is done is up to each MathML renderer. Typically, linebreaking will involve selection of ‘good’ points for insertion of linebreaks between successive arguments of `mrow` elements.

Although MathML does not require linebreaking or specify a particular linebreaking algorithm, it has several features designed to allow such algorithms to produce good results. These include the use of special entities for certain operators, including invisible operators (see Section 3.2.4), or for providing hints related to linebreaking when necessary (see Section 3.2.5), and the ability to use nested `mrow`s to describe sub-expression structure (see below).

`mrow` of one argument

MathML renderers are required to treat an `mrow` element containing exactly one argument as equivalent in all ways to the single argument occurring alone, provided there are no attributes on the `mrow` element’s begin tag. If there are attributes on the `mrow` element’s begin tag, no requirement of equivalence is imposed. This equivalence condition is intended to simplify the implementation of MathML-generating software such as template-based authoring tools. It directly affects the definitions of embellished operator and space-like element and the rules for determining the default value of the `form` attribute of an `mo` element; see sections Section 3.2.4 and Section 3.2.6. See also the discussion of equivalence of MathML expressions in Chapter 7.

3.3.1.3 Proper grouping of sub-expressions using `mrow`

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

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

Although improper grouping will sometimes result in suboptimal renderings, and will often make interpretation other than pure visual rendering difficult or impossible, any grouping of expressions using `mrow` is allowed in MathML syntax; that is, renderers should not assume the rules for proper grouping will be followed.

Precise rule for proper grouping

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

Two adjacent operators (i.e. `mo` elements, possibly embellished), possibly separated by operands (i.e. anything other than operators), should occur in the same `mrow` only when the left operator has an infix or prefix form (perhaps inferred), the right operator has an infix or postfix form, and the operators are listed in the same group of entries in the operator dictionary provided in Appendix D. In all other cases, nested `mrow`s should be used.

When forming a nested `mrow` (during generation of MathML) that includes just one of two successive operators with the forms mentioned above (which mean that either operator could in principle act on the intervening operand or operands), it is necessary to decide which operator acts on those operands directly (or would do so, if they were present). Ideally, this should be determined from the original expression; for example, in conversion from an operator-precedence-based format, it would be the operator with the higher precedence. If this cannot be determined directly from the original expression, the operator that occurs later in the suggested operator dictionary (Appendix D) can be assumed to have a higher precedence for this purpose.

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

(Some of the terminology used in stating the above rule in defined in Section 3.2.4.)

3.3.1.4 Examples

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

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

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

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

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

3.3.2 Fractions (mfrac)

3.3.2.1 Description

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

<mfrac> *numerator* *denominator* </mfrac>

3.3.2.2 Attributes of mfrac

Name	values	default
linethickness	number [v-unit] | thin | medium | thick	1 (rule thickness)
numalign	left | center | right	center
denomalign	left | center | right	center
bevelled	true | false	false

The linethickness attribute indicates the thickness of the horizontal 'fraction bar', or 'rule', typically used to render fractions. A fraction with linethickness="0" renders without the bar, and might be used within binomial coefficients. A linethickness greater than one might be used with nested fractions. These cases are shown below:

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

In general, the value of linethickness can be a number, as a multiplier of the default thickness of the fraction bar (the default thickness is not specified by MathML), or a number with a unit of vertical length (see Section 2.3.4.2), or one of the keywords medium (same as 1), thin (thinner than 1, otherwise up to the renderer), or thick (thicker than 1, otherwise up to the renderer).

The numalign and denomalign attributes control the horizontal alignment of the numerator and denominator respectively. Typically, numerators and denominators are centered, but a very long numerator or denominator might be displayed on several lines and a left alignment might be more appropriate for displaying them.

The bevelled attribute determines whether the fraction is displayed with the numerator above the denominator separated by a horizontal line or whether a diagonal line is used to separate a slightly raised numerator from a slightly lowered denominator. The later form corresponds to the attribute value being true and provides for a more compact form for simple numerator and denominators. An example illustrating the bevelled form is show below:

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

The mfrac element sets displaystyle to false, or if it was already false increments scriptlevel by 1, within *numerator* and *denominator*. These attributes are inherited by every element from its rendering environment, but can be set explicitly only on the mstyle element. (See Section 3.3.4.)

3.3.2.3 Examples

The examples shown above can be represented in MathML as:

```
<mrow>
   <mo> ( </mo>
   <mfrac linethickness="0">
      <mi> a </mi>
      <mi> b </mi>
   </mfrac>
   <mo> ) </mo>
</mrow>
<mfrac linethickness="2">
   <mfrac>
      <mi> a </mi>
      <mi> b </mi>
   </mfrac>
   <mfrac>
      <mi> c </mi>
      <mi> d </mi>
   </mfrac>
</mfrac>
<mfrac>
   <mn> 1 </mn>
   <mrow>
      <msup>
         <mi> x </mi>
         <mn> 3 </mn>
       </msup>
       <mo> + </mo>
       <mfrac>
         <mi> x </mi>
         <mn> 3 </mn>
       </mfrac>
   </mrow>
</mfrac>
<mo> = </mo>
<mfrac bevelled="true">
   <mn> 1 </mn>
   <mrow>
      <msup>
         <mi> x </mi>
         <mn> 3 </mn>
       </msup>
       <mo> + </mo>
       <mfrac>
         <mi> x </mi>
         <mn> 3 </mn>
       </mfrac>
   </mrow>
</mfrac>
```

A more generic example is:

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

3.3.3 Radicals (msqrt, mroot)

3.3.3.1 Description

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

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

The `mroot` element requires exactly 2 arguments. However, `msqrt` accepts any number of arguments; if this number is not 1, its contents are treated as a single 'inferred `mrow`' containing its arguments, as described in Section 3.1.3.

3.3.3.2 Attributes

None (except the attributes allowed for all MathML elements, listed in Section 2.3.5).

The `mroot` element increments `scriptlevel` by 2, and sets `displaystyle` to `false`, within *index*, but leaves both attributes unchanged within *base*. The `msqrt` element leaves both attributes unchanged within all its arguments. These attributes are inherited by every element from its rendering environment, but can be set explicitly only on `mstyle`. (See Section 3.3.4.)

3.3.4 Style Change (mstyle)

3.3.4.1 Description

The `mstyle` element is used to make style changes that affect the rendering of its contents. `mstyle` can be given any attribute accepted by any MathML presentation element provided that the attribute value is inherited, computed or has a default value; presentation element attributes whose values are required are not accepted by the `mstyle` element. In addition `mstyle` can also be given certain special attributes listed below.

The `mstyle` element accepts any number of arguments. If this number is not 1, its contents are treated as a single 'inferred `mrow`' formed from all its arguments, as described in Section 3.1.3.

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

- Some attributes, such as `displaystyle` or `scriptlevel` (explained below), are inherited from the surrounding context when they are not explicitly set. Specifying such an attribute on an `mstyle` element sets the value that will be inherited by its child elements. Unless a child element overrides this inherited value, it will pass it on to its children, and they will pass it to their children, and so on. But if a child element does override it, either by an explicit attribute setting or automatically (as is common for `scriptlevel`), the new (overriding) value will be passed on to that element's children, and then to their children, etc, until it is again overridden.

- Other attributes, such as linethickness on mfrac, have default values that are not normally inherited. That is, if the linethickness attribute is not set on the begin tag of an mfrac element, it will normally use the default value of 1, even if it was contained in a larger mfrac element that set this attribute to a different value. For attributes like this, specifying a value with an mstyle element has the effect of changing the default value for all elements within its scope. The net effect is that setting the attribute value with mstyle propagates the change to all the elements it contains directly or indirectly, except for the individual elements on which the value is overridden. Unlike in the case of inherited attributes, elements that explicitly override this attribute have no effect on this attribute's value in their children.
- Another group of attributes, such as stretchy and form, are computed from operator dictionary information, position in the enclosing mrow, and other similar data. For these attributes, a value specified by an enclosing mstyle overrides the value that would normally be computed.

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

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

There is one exceptional element, mpadded, whose attributes cannot be set with mstyle. When the attributes width, height and depth are specified on an mstyle element, they apply only to the mspace element. Similarly, when lspace is set with mstyle, it applies only to the mo element.

3.3.4.2 Attributes

As stated above, mstyle accepts all attributes of all MathML presentation elements which do not have required values. That is, all attributes which have an explicit default value or a default value which is inherited or computed are accepted by the mstyle element. Additionally, mstyle can be given the following special attributes that are implicitly inherited by every MathML element as part of its rendering environment:

Name	values	default
scriptlevel	['+' | '-'] unsigned-integer	inherited
displaystyle	true | false	inherited
scriptsizemultiplier	number	0.71
scriptminsize	number v-unit	8pt
color	#rgb | #rrggbb | html-color-name	inherited
background	#rgb | #rrggbb | transparent | html-color-name	transparent
veryverythinmathspace	number h-unit	0.0555556em
verythinmathspace	number h-unit	0.111111em
thinmathspace	number h-unit	0.166667em
mediummathspace	number h-unit	0.222222em
thickmathspace	number h-unit	0.277778em
verythickmathspace	number h-unit	0.333333em
veryverythickmathspace	number h-unit	0.388889em

scriptlevel and displaystyle

MathML uses two attributes, displaystyle and scriptlevel, to control orthogonal presentation features that TEX encodes into one style attribute with values \displaystyle, \textstyle, \scriptstyle, and \scriptscriptstyle. The corresponding values of displaystyle and scriptlevel for those TEX styles would be true and 0, false and 0, false and 1, and false and 2, respectively.

The main effect of the displaystyle attribute is that it determines the effect of other attributes such as the largeop and movablescripts attributes of mo. The main effect of the scriptlevel attribute 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.) More sophisticated renderers may also choose to use these attributes in other ways, such as rendering expressions with displaystyle=false in a more vertically compressed manner.

These attributes are given initial values for the outermost expression of an instance of MathML based on its rendering environment. A short list of layout schemata described below modify these values for some of their sub-expressions. Otherwise, values are determined by inheritance whenever they are not directly specified on a given element's start tag.

For an instance of MathML embedded in a textual data format (such as HTML) in 'display' mode, i.e. in place of a paragraph, displaystyle = true and scriptlevel = 0 for the outermost expression of the embedded MathML; if the MathML is embedded in 'inline' mode, i.e. in place of a character, displaystyle = false and scriptlevel = 0 for the outermost expression. See Chapter 7 for further discussion of the distinction between 'display' and 'inline' embedding of MathML and how this can be specified in particular instances. In general, a MathML renderer may determine these initial values in whatever manner is appropriate for the location and context of the specific instance of MathML it is rendering, or if it has no way to determine this, based on the way it is most likely to be used; as a last resort it is suggested that it use the most generic values displaystyle = "true" and scriptlevel = "0".

The MathML layout schemata that typically display some of their arguments in smaller type or with less vertical spacing, namely the elements for scripts, fractions, radicals, and tables or matrices, set displaystyle to false, and in some cases increase scriptlevel, for those arguments. The new values are inherited by all sub-expressions within those arguments, unless they are overridden.

The specific rules by which each element modifies displaystyle and/or scriptlevel are given in the specification for each element that does so; the complete list of elements that modify either attribute are: the 'scripting' elements msub, msup, msubsup, munder, mover, munderover, and mmultiscripts; and the elements mfrac, mroot, and mtable.

When mstyle is given a scriptlevel attribute with no sign, it sets the value of scriptlevel within its contents to the value given, which must be a nonnegative integer. When the attribute value consists of a sign followed by an integer, the value of scriptlevel is incremented (for '+') or decremented (for '-') by the amount given. The incremental syntax for this attribute is an exception to the general rules for setting inherited attributes using mstyle, and is not allowed by any other attribute on mstyle.

Whenever the scriptlevel is changed, either automatically or by being explicitly incremented, decremented, or set, the current font size is multiplied by the value of scriptsizemultiplier to the power of the change in scriptlevel. For example, if scriptlevel is increased by 2, the font size is multiplied by scriptsizemultiplier twice in succession; if scriptlevel is explicitly set to 2 when it had been 3, the font size is divided by scriptsizemultiplier.

The default value of scriptsizemultiplier is less than one (in fact, it is approximately the square root of 1/2), resulting in a smaller font size with increasing scriptlevel. To prevent scripts from becoming unreadably small, the font size is never allowed to go below the value of scriptminsize as a result of a change to scriptlevel, though it can be set to a lower value using the fontsize attribute (Section 3.2.1) on mstyle or on token elements. If a change to scriptlevel would cause the font size to become lower than scriptminsize using the above formula, the font size is instead set equal to scriptminsize within the sub-expression for which scriptlevel was changed.

In the syntax for scriptminsize, v-unit represents a unit of vertical length (as described in Section 2.3.4.2). The most common unit for specifying font sizes in typesetting is pt (points).

Explicit changes to the fontsize attribute have no effect on the value of scriptlevel.

Further details on scriptlevel for renderers

For MathML renderers that support CSS1 style sheets, or some other analogous style sheet mechanism, absolute or relative changes to fontsize (or other attributes) may occur implicitly on any element in response to a style sheet. Changes to fontsize of this kind also have no effect on scriptlevel. A style sheet-induced change to fontsize overrides scriptminsize in the same way as for an explicit change to fontsize in the element's begin tag (discussed above), whether it is specified in the style sheet as an absolute or a relative change. (However, any subsequent scriptlevel-induced change to fontsize will still be affected by it.) As is required for inherited attributes in CSS1, the style sheet-modified fontsize is inherited by child elements.

If the same element is subject to both a style sheet-induced and an automatic (scriptlevel-related) change to its own fontsize, the scriptlevel-related change is done first - in fact, in the simplest implementation of the element-specific rules for scriptlevel, this change would be done by the element's parent as part of producing the rendering properties it passes to the given element, since it is the parent element that knows whether scriptlevel should be changed for each of its child elements.

If the element's own fontsize is changed by a style sheet and it also changes scriptlevel (and thus fontsize) for one of its children, the style sheet-induced change is done first, followed by the change inherited by that child. If more than one child's scriptlevel is changed, the change inherited by each child has no effect on the other children. (As a mnemonic rule that applies to a 'parse tree' of elements and their children, style sheet-induced changes to fontsize can be associated to nodes of the tree, i.e. to MathML elements, and scriptlevel-related changes can be associated to the edges between parent and child elements; then the order of the associated changes corresponds to the order of nodes and edges in each path down the tree.) For general information on the relative order of processing of properties set by style sheets versus by attributes, see the appropriate subsection of CSS-compatible attributes in Section 2.3.4.3.

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

Note that scriptlevel can, in principle, attain any integral value by being decremented sufficiently, even though it can only be explicitly set to nonnegative values. Negative values of scriptlevel generated in this way are legal and should work as described, generating font sizes larger than those of the surrounding expression. Since scriptlevel is initially 0 and never decreases automatically, it will always be nonnegative unless it is decremented past 0 using mstyle.

Explicit decrements of scriptlevel after the font size has been limited by scriptminsize as described above would produce undesirable results. This might occur, for example, in a representation of a continued fraction, in which the scriptlevel was decremented for part of the denominator back to its value for the fraction as a whole, if the continued fraction itself was located in a place that had a high scriptlevel. To prevent this problem, MathML renderers should, when decrementing scriptlevel, use as the initial font size the value the font size would have had if it had never been limited by scriptminsize. They should not, however, ignore the effects of explicit settings of fontsize, even to values below scriptminsize.

Since MathML renderers may be unable to make use of arbitrary font sizes with good results, they may wish to modify the mapping from scriptlevel to fontsize to produce better renderings in their judgment. In particular, if fontsizes have to be rounded to available values, or limited to values within a range, the details of how this is done are up to the renderer. Renderers should, however, ensure that a series of incremental changes to scriptlevel resulting in its return to the same value for some sub-expression that it had in a surrounding expression results in the same fontsize for that sub-expression as for the surrounding expression.

Color and background attributes

The color attribute controls the color in which the content of tokens is rendered. Additionally, when inherited from mstyle or from a MathML expression's rendering environment, it controls the color of all other drawing by MathML elements, including the lines or radical signs that can be drawn by mfrac, mtable, or msqrt.

Note that the background attribute, though not inherited, has the default value 'transparent' (as in CSS1), which effectively allows an element's parent to control its background.

The values of color and background can be specified as a string consisting of '#' followed without intervening whitespace by either 1-digit or 2-digit hexadecimal values for the red, green, and blue components, respectively, of the desired color, with the same number of digits used for each component (or as the keyword 'transparent' for background). The hexadecimal digits are not case-sensitive. The possible 1-digit values range from 0 (component not present) to F (component fully present), and the possible 2-digit values range from 00 (component not present) to FF (component fully present), with the 1-digit value *x* being equivalent to the 2-digit value *xx* (rather than *x0*). % *x*0 would be a more strictly correct notation, but renders terribly in some browsers.

These attributes can also be specified as an html-color-name, which is defined in the following subsection.

CSS compatibility of color attributes

The color syntax described above is a subset of the syntax of the color and background-color properties of CSS1. (The background-color syntax is in turn a subset of the full CSS1 background property syntax, which also permits specification of (for example) background images with optional repeats. The more general attribute name background is used in MathML to facilitate possible extensions to the attribute's scope in future versions of MathML.)

Color values on either attribute can also be specified as an `html-color-name`, that is, as one of the color-name keywords defined in [HTML4.0]. The list of allowed color names includes most of the commonest English color words, though not `orange`, `brown`, or `pink`, and also includes a number of less-common color words; see the reference for the complete list and the equivalent RGB values. Note that the color name keywords are not case-sensitive, unlike most keywords in MathML attribute values. (The same color name keywords are defined for the CSS1 `color` property, but with unspecified RGB values. See also Section 2.3.4.3.)

Precise background region not specified

The suggested MathML visual rendering rules do not define the precise extent of the region whose background is affected by using the `background` attribute on `mstyle`, except that, when `mstyle`'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 `mstyle`, 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.

Meaning of named mathspaces

The spacing between operators is often one of a small number of potential values. MathML names these values and allows their values to be changed. Because the default values for spacing around operators that are given in the operator dictionary Appendix D are defined using these named spaces, changing their values will produce tighter or looser spacing. These values can be used anywhere a `h-unit` or `v-unit` unit is allowed Section 2.3.4.2.

The predefined `namedspaces` are: `veryverythinmathspace`, `verythinmathspace`, `thinmathspace`, `mediummathspace`, `thickmathspace`, `verythick-mathspace`, or `veryverythickmathspace`. The default values of `veryverythinmathspace`... `veryverythickmathspace` are 1/18em...7/18em, respectively.

3.3.4.3 Examples

The example of limiting the stretchiness of a parenthesis shown in the section on <mo>,

```
<mrow>
   <mo maxsize="1"> ( </mo>
   <mfrac> <mi> a </mi> <mi> b </mi> </mfrac>
   <mo maxsize="1"> ) </mo>
</mrow>
```

can be rewritten using `mstyle` as:

```
<mstyle maxsize="1">
   <mrow>
      <mo> ( </mo>
      <mfrac> <mi> a </mi> <mi> b </mi> </mfrac>
      <mo> ) </mo>
   </mrow>
</mstyle>
```

3.3.5 Error Message (`merror`)

3.3.5.1 Description

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

`merror` accepts any number of arguments; if this number is not 1, its contents are treated as a single ‘inferred `mrow`’ as described in Section 3.1.3.

The intent of this element is to provide a standard way for programs that *generate* MathML from other input to report syntax errors in their input. Since it is anticipated that preprocessors that parse input syntaxes designed for easy hand entry will be developed to generate MathML, it is important that they have the ability to indicate that a syntax error occurred at a certain point. See Section 7.2.2.

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

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

3.3.5.2 Attributes

None (except the attributes allowed for all MathML elements, listed in Section 2.3.5).

3.3.5.3 Example

If a MathML syntax-checking preprocessor received the input

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

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

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

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

3.3.6 Adjust Space Around Content (`mpadded`)

3.3.6.1 Description

An `mpadded` element renders the same as its content, but with its overall size and other dimensions (such as baseline position) modified according to its attributes. The `mpadded` element does not rescale (stretch or shrink) its content; its only effect is to modify the apparent size and position of the 'bounding box' around its content, so as to affect the relative position of the content with respect to the surrounding elements. The name of the element reflects the use of `mpadded` to effectively add 'padding', or extra space, around its content. If the 'padding' is negative, it is possible for the content of `mpadded` to be rendered outside the `mpadded` element's bounding box; see below for warnings about several potential pitfalls of this effect.

The `mpadded` element accepts any number of arguments; if this number is not 1, its contents are treated as a single 'inferred `mrow`' as described in Section 3.1.3.

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

3.3.6.2 Attributes

Name	values
width	[+ | -] *unsigned-number* (% [*pseudo-unit*] | *pseudo-unit* | *h-unit* | namedspace)
lspace	[+ | -] *unsigned-number* (% [*pseudo-unit*] | *pseudo-unit* | *h-unit*)
height	[+ | -] *unsigned-number* (% [*pseudo-unit*] | *pseudo-unit* | *v-unit*)
depth	[+ | -] *unsigned-number* (% [*pseudo-unit*] | *pseudo-unit* | *v-unit*)

(The *pseudo-unit* syntax symbol is described below.)

These attributes modify the dimensions of the 'bounding box' of the `mpadded` element. The dimensions (which have the same names as the attributes) are defined in the next subsection. Depending on the format of the attribute value, a dimension may be set to a new value, or to an incremented or decremented version of the content's corresponding dimension. Values may be specified as multiples or percentages of any of the dimensions of the normal rendering of the element's content (using so-called 'pseudo-units'), or they can be set directly using standard units Section 2.3.4.2.

If an attribute value begins with a + or − sign, it specifies an increment or decrement of the corresponding dimension by the following length value (interpreted as explained below). Otherwise, the corresponding dimension is set directly to the following length value. Note that the + and − do not mean that the following value is positive or negative, even when an explicit length unit (*h-unit* or *v-unit*) is given. In particular, these attributes cannot directly set a dimension to a negative value.

Length values (after the optional sign, which is not part of the length value) can be specified in several formats. Each format begins with an *unsigned-number*, which may be followed by a % sign and an optional 'pseudo-unit' (denoted by *pseudo-unit* in the attribute syntaxes above), by a pseudo-unit alone, or by one of the length units (denoted by *h-unit* or *v-unit*) specified in Section 2.3.4.2, not including %. The possible pseudo-units are the keywords `width`, `lspace`, `height`, and `depth`; they each represent the length of the same-named dimension of the `mpadded` element's content (not of the `mpadded` element itself). The lengths represented by *h-unit* or *v-unit* are described in Section 2.3.4.2.

In any of these formats, the length value specified is the product of the specified number the length represented by the unit or pseudo-unit, and multiplied by 0.01 if % is given. If no pseudo-unit is given after %, the one with the same name as the attribute being specified is assumed.

Some examples of attribute formats using pseudo-units (explicit or default) are as follows: `depth="100% height"` and `depth="1.0 height"` both set the depth of the `mpadded` element to the height of its content. `depth="105%"` sets the depth to 1.05 times the content's depth, and either `depth="+100%"` or `depth="200%"` sets the depth to twice the content's depth.

Dimensions that would be positive if the content was rendered normally cannot be made negative using `mpadded`; a positive dimension is set to 0 if it would otherwise become negative. Dimensions that are initially 0 can be made negative, but this should generally be avoided. See the warnings below on the use of negative spacing for 'tweaking' or conveying meaning.

The rules given above imply that all of the following attribute settings have the same effect, which is to leave the content's dimensions unchanged:

```
<mpadded width="+0em"> ... </mpadded>
<mpadded width="+0%"> ... </mpadded>
<mpadded width="-0em"> ... </mpadded>
<mpadded width="- 0 height"> ... </mpadded>
<mpadded width="100%"> ... </mpadded>
<mpadded width="100% width"> ... </mpadded>
<mpadded width="1 width"> ... </mpadded>
<mpadded width="1.0 width"> ... </mpadded>
<mpadded> ... </mpadded>
```

3.3.6.3 Meanings of dimension attributes

See Appendix F for further information about some of the typesetting terms used here.

The width attribute refers to the overall horizontal width of a bounding box. By default (i.e. when lspace is not modified), the bounding box of the content of an mpadded element should be rendered flush with the left edge of the mpadded element's bounding box. Thus, increasing width alone effectively adds space on the right edge of the box.

The lspace attribute refers to the amount of space between the left edge of a bounding box and where the rendering of its contents' bounding box actually begins. Unlike the other dimensions, lspace does not correspond to a real property of a bounding box, but exists only transiently during the computations done by each instance of mpadded. It is provided so that there is a way to add space on the left edge of a bounding box.

The rationale behind using width and lspace to control horizontal padding instead of more symmetric attributes, such as a hypothetical rspace and lspace, is that it is desirable to have a 'width' pseudo unit, in part because 'width' is an actual property of a bounding box.

The height attribute refers to the amount of vertical space between the baseline (the line along the bottom of most letter glyphs in normal text rendering) and the top of the bounding box.

The depth attribute refers to the amount of vertical space between the bottom of the bounding box and the baseline.

MathML renderers should ensure that, except for the effects of the attributes, relative spacing between the contents of mpadded and surrounding MathML elements is not modified by replacing an mpadded element with an mrow element with the same content. This holds even if linebreaking occurs within the mpadded element. However, if an mpadded element with non-default attribute values is subjected to linebreaking, MathML does not define how its attributes or rendering interact with the linebreaking algorithm.

3.3.6.4 Warning: nonportability of 'tweaking'

A likely temptation for the use of the mpadded and mspace elements (and perhaps also mphantom and mtext) will be for an author to improve the spacing generated by a specific renderer by slightly modifying it in specific expressions, i.e. to 'tweak' the rendering.

Authors are strongly warned that *different MathML renderers may use different spacing rules* for computing the relative positions of rendered symbols in expressions that have no explicit modifications to their spacing; if renderer B improves upon renderer A's spacing rules, explicit spacing added to improve the output quality of renderer A may produce very poor results in renderer B, very likely worse than without any 'tweaking' 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 implementors of MathML renderers might be still more productive, in the long run.

3.3.6.5 Warning: spacing should not be used to convey meaning

MathML elements that permit 'negative spacing', namely mspace, mpadded, and mtext, 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);
- 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.

If such 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.

For example, the MathML expression

```
<mrow>
  <mpadded width="0"> <mi> C </mi> </mpadded>
  <mspace width="0.3em"/>
  <mtext> | </mtext>
</mrow>
```

forms an overstruck symbol in violation of the policy stated above; it might be intended to represent the set of complex numbers for a MathML renderer that lacks support for the standard symbol used for this purpose. This kind of construct should always be avoided in MathML, for the reasons stated above; indeed, it should never be necessary for standard symbols, since a MathML renderer with no better method of rendering them is free to use overstriking internally, so that it can still support general MathML input.

However, if for whatever reason such a construct is used in MathML, it should always be enclosed in a `semantics` element such as

```
<semantics>
  <mrow>
    <mpadded width="0"> <mi> C </mi> </mpadded>
    <mspace width="0.3em"/>
    <mtext> | </mtext>
  </mrow>
  <annotation-xml encoding="MathML-Presentation">
    <mi> &Copf; </mi>
  </annotation-xml>
</semantics>
```

which provides an alternative, standard encoding for the desired symbol, which is much more easily interpreted than the construct using negative spacing. (The alternative encoding in this example uses MathML presentation elements; the content elements described in Chapter 4 should also be considered.)

(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 `fontweight`) used to distinguish one variable from another.)

3.3.7 Making Content Invisible (`mphantom`)

3.3.7.1 Description

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

The `mphantom` element accepts any number of arguments; if this number is not 1, its contents are treated as a single 'inferred `mrow`' formed from all its arguments, as described in Section 3.1.3.

It is suggested that audio renderers render `mphantom` elements in an analogous way for their medium, by rendering them as silence of the same duration as the normal rendering of their contents.

3.3.7.2 Attributes

None (except the attributes allowed for all MathML elements, listed in Section 2.3.5).

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

MathML renderers should ensure that the relative spacing between the contents of an mphantom element and the surrounding MathML elements is the same as it would be if the mphantom element were replaced by an mrow element with the same content. This holds even if linebreaking occurs within the mphantom element.

For the above reason, mphantom is *not* considered space-like (Section 3.2.6) unless its content is space-like, since the suggested rendering rules for operators are affected by whether nearby elements are space-like. Even so, the warning about the legal grouping of space-like elements may apply to uses of mphantom.

There is one situation where the preceding rule for rendering an mphantom may not give the desired effect. When an mphantom is wrapped around a subsequence of the arguments of an mrow, the default determination of the form attribute for an mo element within the subsequence can change. (See the default value of the form attribute described in Section 3.2.4.) It may be necessary to add an explicit form attribute to such an mo in these cases. This is illustrated in the following example.

3.3.7.3 Examples

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

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

This would render as something like

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

rather than as

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

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

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

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

3.3.8 Content Inside Pair of Fences (`mfenced`)

3.3.8.1 Description

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

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

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

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

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

Individual fences or separators are represented using `mo` elements, as described in Section 3.2.4. Thus, any `mfenced` element is completely equivalent to an expanded form described below; either form can be used in MathML, at the convenience of an author or of a MathML-generating program. A MathML renderer is required to render either of these forms in exactly the same way.

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

3.3.8.2 Attributes

Name	values	default
open	string	(
close	string	)
separators	character *	,

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

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

The opening-fence and closing-fence are arbitrary strings. (Since they are used as the content of mo elements, any whitespace they contain will be trimmed and collapsed as described in Section 2.3.6.)

The value of separators is a sequence of zero or more separator characters (or entity references), optionally separated by whitespace. Each sep#i consists of exactly one character or entity reference. Thus, separators=",;" is equivalent to separators=" , ; ".

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

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

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

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

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

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

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

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

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

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

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

Note that the equivalent expanded forms shown above include attributes on the `mo` elements that identify them as fences or separators. Since the most common choices of fences and separators already occur in the operator dictionary with those attributes, authors would not normally need to specify those attributes explicitly when using the expanded form directly. Also, the rules for the default `form` attribute (Section 3.2.4) cause the opening and closing fences to be effectively given the values `form`="prefix" and `form`="postfix" respectively, and the separators to be given the value `form`="infix".

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

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

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

3.3.8.3 Examples

$(a+b)$

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

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

[0,1)

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

$f(x,y)$

```
<mrow>
  <mi> f </mi>
  <mo> &ApplyFunction; </mo>
  <mfenced>
    <mi> x </mi>
    <mi> y </mi>
  </mfenced>
</mrow>
```

3.3.9 Enclose Content Inside Notation (menclose)

3.3.9.1 Description

The menclose element renders its content inside the enclosing notation specified by its notation attribute. menclose accepts any number of arguments; if this number is not 1, its contents are treated as a single 'inferred mrow' containing its arguments, as described in Section 3.1.3.

3.3.9.2 Attributes

Name	values	default
notation	longdiv | actuarial | radical	longdiv

When notation has the value longdiv, the contents are drawn enclosed by a long division symbol. A complete example of long division is accomplished by also using mtable and malign. When notation is specified as actuarial, the contents are drawn enclosed by an actuarial symbol. The case of notation=radical is equivalent to the msqrt schema.

3.3.9.3 Examples

The following markup might be used to encode an elementary US-style long division problem.

```
<mtable columnspacing='0' rowspacing='0'>
<mtr>
  <mtd></mtd>
  <mtd columnalign='right'><mn>10</mn></mtd>
</mtr>
<mtr>
  <mtd columnalign='right'><mn>131</mn></mtd>
  <mtd columnalign='right'>
    <menclose notation='longdiv'><mn>1413</mn></menclose>
  </mtd>
</mtr>
<mtr>
  <mtd></mtd>
  <mtd columnalign='right'>
    <mrow>
    <munder>
      <mn>131</mn>
      <mo>&UnderBar;</mo>
    </munder>
    <mphantom><mn>3</mn></mphantom>
    </mrow>
  </mtd>
</mtr>
<mtr>
  <mtd></mtd>
  <mtd columnalign='right'><mn>103</mn></mtd>
</mtr>
</mtable>
```

This might be rendered roughly as:

$$\begin{array}{rr} & 10 \\ 131 & \overline{)1413} \\ & \underline{131} \\ & 103 \end{array}$$

An example of using `menclose` for actuarial notation is

```
<msub>
  <mi>a</mi>
  <mrow>
    <menclose notation='actuarial'>
      <mi>n</mi>
    </menclose>
    <mo>&it;</mo>
    <mi>i</mi>
  </mrow>
</msub>
```

which renders roughly as

$$\frac{a}{n} | \quad i$$

3.4 Script and Limit Schemata

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

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

Rendering of scripts is affected by the `scriptlevel` and `displaystyle` attributes, which are part of the environment inherited by the rendering process of every MathML expression, and are described under `mstyle` (Section 3.3.4). These attributes cannot be given explicitly on a scripting element, but can be specified on the start tag of a surrounding `mstyle` element if desired.

MathML also provides an element for attachment of tensor indices. Tensor indices are distinct from ordinary subscripts and superscripts in that they must align in vertical columns. Tensor indices can also occur in prescript positions.

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

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

3.4.1 Subscript (`msub`)

3.4.1.1 Description

The syntax for the `msub` element is:

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

3.4.1.2 Attributes

Name	values	default
subscriptshift	number v-unit	automatic (typical unit is ex)

The `subscriptshift` attribute specifies the minimum amount to shift the baseline of *subscript* down.

v-unit represents a unit of vertical length (see Section 2.3.4.2).

The `msub` element increments `scriptlevel` by 1, and sets `displaystyle` to `false`, within *subscript*, but leaves both attributes unchanged within *base*. (These attributes are inherited by every element through its rendering environment, but can be set explicitly only on `mstyle`; see Section 3.3.4.)

3.4.2 Superscript (`msup`)

3.4.2.1 Description

The syntax for the `msup` element is:

`<msub>` *base* *superscript* `</msub>`

3.4.2.2 Attributes

Name	values	default
superscriptshift	number v-unit	automatic (typical unit is ex)

The `superscriptshift` attribute specifies the minimum amount to shift the baseline of *superscript* up.

v-unit represents a unit of vertical length (see Section 2.3.4.2).

The `msup` element increments `scriptlevel` by 1, and sets `displaystyle` to `false`, within *superscript*, but leaves both attributes unchanged within *base*. (These attributes are inherited by every element through its rendering environment, but can be set explicitly only on `mstyle`; see Section 3.3.4.)

3.4.3 Subscript-superscript Pair (`msubsup`)

3.4.3.1 Description

The `msubsup` element is used to attach both a subscript and superscript to a base expression. Note that both scripts are positioned tight against the base: ${x_1}^2$ versus x_1^2.

The syntax for the `msubsup` element is:

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

3.4.3.2 Attributes

Name	values	default
subscriptshift	number v-unit	automatic (typical unit is ex)
superscriptshift	number v-unit	automatic (typical unit is ex)

The `subscriptshift` attribute specifies the minimum amount to shift the baseline of *subscript* down. The `superscriptshift` attribute specifies the minimum amount to shift the baseline of *superscript* up.

v-unit represents a unit of vertical length (see Section 2.3.4.2).

The `msubsup` element increments `scriptlevel` by 1, and sets `displaystyle` to `false`, within *subscript* and *superscript*, but leaves both attributes unchanged within *base*. (These attributes are inherited by every element through its rendering environment, but can be set explicitly only on `mstyle`; see Section 3.3.4.)

3.4.3.3 Examples

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

$$\int_0^1 e^x \, dx$$

would be represented as

```
<mrow>
  <msubsup>
    <mo> &int; </mo>
    <mn> 0 </mn>
    <mn> 1 </mn>
  </msubsup>
  <mrow>
    <msup>
      <mi> &ExponentialE; </mi>
      <mi> x </mi>
    </msup>
    <mo> &InvisibleTimes; </mo>
    <mrow>
      <mo> &DifferentialD; </mo>
      <mi> x </mi>
    </mrow>
  </mrow>
</mrow>
```

3.4.4 Underscript (`munder`)

3.4.4.1 Description

The syntax for the `munder` element is:

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

3.4.4.2 Attributes

Name	values	default
accentunder	true | false	automatic

The `accentunder` attribute controls whether *underscript* is drawn as an ‘accent’ or as a limit. The main difference between an accent and a limit is that the limit is reduced in size whereas an accent is the same size as the base. A second difference is that the accent is drawn closer to the base.

The default value of `accentunder` is false, unless *underscript* is an `mo` element or an embellished operator (see Section 3.2.4). If *underscript* is an `mo` element, the value of its `accent` attribute is used as the default value of `accentunder`. If *underscript* is an embellished operator, the `accent` attribute of the `mo` element at its core is used as the default value. As with all attributes, an explicitly given value overrides the default.

Here is an example (accent versus underscript): $\underbrace{x+y+z}$ versus $\underbrace{x+y+z}$. The MathML representation for this example is shown below.

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

Within *underscript*, `munder` always sets `displaystyle` to `false`, but increments `scriptlevel` by 1 only when `accentunder` is `false`. Within *base*, it always leaves both attributes unchanged. (These attributes are inherited by every element through its rendering environment, but can be set explicitly only on `mstyle`; see Section 3.3.4.)

3.4.4.3 Examples

The MathML representation for the example shown above is:

```
<mrow>
  <munder accentunder="true">
    <mrow>
      <mi> x </mi>
      <mo> + </mo>
      <mi> y </mi>
      <mo> + </mo>
      <mi> z </mi>
    </mrow>
    <mo> &UnderBrace; </mo>
  </munder>
  <mtext> versus </mtext>
  <munder accentunder="false">
    <mrow>
      <mi> x </mi>
      <mo> + </mo>
      <mi> y </mi>
      <mo> + </mo>
      <mi> z </mi>
    </mrow>
    <mo> &UnderBrace; </mo>
  </munder>
</mrow>
```

3.4.5 Overscript (`mover`)

3.4.5.1 Description

The syntax for the `mover` element is:

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

3.4.5.2 Attributes

Name	values	default
accent	true | false	automatic

The `accent` attribute controls whether *overscript* is drawn as an ‘accent’ (diacritical mark) or as a limit. The main difference between an accent and a limit is that the limit is reduced in size whereas an accent is the same size as the base. A second difference is that the accent is drawn closer to the base. This is shown below (accent versus limit): $\hat{x}$ versus $\overset{\hat{}}{x}$.

These differences also apply to 'mathematical accents' such as bars over expressions: $\overbrace{x+y+z}$ versus $\overbrace{x+y+z}$. The MathML representation for each of these examples is shown below.

The default value of *accent* is false, unless *overscript* is an `mo` element or an embellished operator (see Section 3.2.4). If *overscript* is an `mo` element, the value of its `accent` attribute is used as the default value of `accent` for `mover`. If *overscript* is an embellished operator, the `accent` attribute of the `mo` element at its core is used as the default value.

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

Within *overscript*, `mover` always sets `displaystyle` to `false`, but increments `scriptlevel` by 1 only when `accent` is `false`. Within *base*, it always leaves both attributes unchanged. (These attributes are inherited by every element through its rendering environment, but can be set explicitly only on `mstyle`; see Section 3.3.4.)

3.4.5.3 Examples

The MathML representation for the examples shown above is:

```
<mrow>
  <mover accent="true">
    <mi> x </mi>
    <mo> &Hat; </mo>
  </mover>
  <mtext> versus </mtext>
  <mover accent="false">
    <mi> x </mi>
    <mo> &Hat; </mo>
  </mover>
</mrow>
```

```
<mrow>
  <mover accent="true">
    <mrow>
      <mi> x </mi>
      <mo> + </mo>
      <mi> y </mi>
      <mo> + </mo>
      <mi> z </mi>
    </mrow>
    <mo> &OverBar; </mo>
  </mover>
  <mtext> versus </mtext>
  <mover accent="false">
    <mrow>
      <mi> x </mi>
      <mo> + </mo>
      <mi> y </mi>
      <mo> + </mo>
      <mi> z </mi>
    </mrow>
    <mo> &OverBar; </mo>
  </mover>
</mrow>
```

3.4.6 Underscript-overscript Pair (`munderover`)

3.4.6.1 Description

The syntax for the `munderover` element is:

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

3.4.6.2 Attributes

Name	values	default
accent	true | false	automatic
accentunder	true | false	automatic

The `munderover` element is used so that the underscript and overscript are vertically spaced equally in relation to the base and so that they follow the slant of the base as in the second expression shown below:

$$\int\limits_0^\infty$$

versus

$$\int_0^\infty$$

The MathML representation for this example is shown below.

The difference in the vertical spacing is too small to be noticed on a low resolution display at a normal font size, but is noticeable on a higher resolution device such as a printer and when using large font sizes. In addition to the visual differences, attaching both the underscript and overscript to the same base more accurately reflects the semantics of the expression.

The `accent` and `accentunder` attributes have the same effect as the attributes with the same names on `mover` (Section 3.4.5) and `munder` (Section 3.4.4), respectively. Their default values are also computed in the same manner as described for those elements, with the default value of `accent` depending on *overscript* and the default value of `accentunder` depending on *underscript*.

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

Within *underscript*, `munderover` always sets `displaystyle` to `false`, but increments `scriptlevel` by 1 only when `accentunder` is `false`. Within *overscript*, `munderover` always sets `displaystyle` to `false`, but increments `scriptlevel` by 1 only when `accent` is `false`. Within *base*, it always leaves both attributes unchanged. (These attributes are inherited by every element through its rendering environment, but can be set explicitly only on `mstyle`; see Section 3.3.4).

3.4.6.3 Examples

The MathML representation for the example shown above with the first expression made using separate `munder` and `mover` elements, and the second one using an `munderover` element, is:

```
<mrow>
  <mover>
    <munder>
      <mo> &int; </mo>
      <mn> 0 </mn>
    </munder>
    <mi> &infin; </mi>
  </mover>
  <mtext> versus </mtext>
  <munderover>
    <mo> &int; </mo>
    <mn> 0 </mn>
    <mi> &infin; </mi>
  </munderover>
</mrow>
```

3.4.7 Prescripts and Tensor Indices (`mmultiscripts`)

3.4.7.1 Description

The syntax for the `mmultiscripts` element is:

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

Presubscripts and tensor notations are represented by a single element, `mmultiscripts`. This element allows the representation of any number of vertically-aligned pairs of subscripts and superscripts, attached to one base expression. It supports both postscripts (to the right of the base in visual notation) and prescripts (to the left of the base in visual notation). Missing scripts can be represented by the empty element `none`.

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

The argument sequence consists of the base followed by zero or more pairs of vertically-aligned subscripts and superscripts (in that order) that represent all of the postscripts. This list is optionally followed by an empty element `mprescripts` and a list of zero or more pairs of vertically-aligned presubscripts and presuperscripts that represent all of the prescripts. The pair lists for postscripts and prescripts are given in a left-to-right order. If no subscript or superscript should be rendered in a given position, then the empty element `none` should be used in that position.

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

The empty elements `mprescripts` and `none` are only allowed as direct sub-expressions of `mmultiscripts`.

3.4.7.2 Attributes

Same as the attributes of `msubsup`.

The `mmultiscripts` element increments `scriptlevel` by 1, and sets `displaystyle` to `false`, within each of its arguments except *base*, but leaves both attributes unchanged within *base*. (These attributes are inherited by every element through its rendering environment, but can be set explicitly only on `mstyle`; see Section 3.3.4.)

3.4.7.3 Examples

Two examples of the use of `mmultiscripts` are:

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

```
<mrow>
  <mmultiscripts>
    <mi> F </mi>
    <mn> 1 </mn>
    <none/>
    <mprescripts/>
    <mn> 0 </mn>
    <none/>
  </mmultiscripts>
  <mo> &ApplyFunction; </mo>
  <mrow>
    <mo> ( </mo>
    <mrow>
      <mo> ; </mo>
      <mi> a </mi>
      <mo> ; </mo>
      <mi> z </mi>
    </mrow>
    <mo> ) </mo>
  </mrow>
</mrow>
```

$R_{i\ kl}^{\ j}$ (where k and l are different indices)

```
<mmultiscripts>
  <mi> R </mi>
  <mi> i </mi>
  <none/>
  <none/>
  <mi> j </mi>
  <mi> k </mi>
  <none/>
  <mi> l </mi>
  <none/>
</mmultiscripts>
```

3.5 Tables and Matrices

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

The `mlabeledtr` element represents a labeled row of a table and can be used for numbered equations. `mlabeledtr` first child is the label. A label is somewhat special in that it is not considered an expression in the matrix and is not counted when determining the number of columns in that row.

3.5.1 Table or Matrix (`mtable`)

3.5.1.1 Description

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

In MathML 1.x, the `mtable` element could infer `mtr` elements around its arguments, and the `mtr` element could infer `mtd` elements. In other words, if some argument to an `mtable` was not an `mtr` element, a MathML application was to assume a row with a single column (i.e. the argument was effectively wrapped with an inferred `mtr`). Similarly, if some argument to a (possibly inferred) `mtr` element was not an `mtd` element, that argument was to be treated as a table entry by wrapping it with an inferred `mtd` element.

In MathML 2.0, `mtr` and `mtd` elements are required, and may no longer be inferred. However, for backward compatibility renderers may wish to continue supporting inferred `mtr` and `mtd` elements. In this case, however, renderers should make an effort to notify users that inferred `mtr` and `mtd` elements are not valid in MathML 2.0.

Table rows that have fewer columns than other rows of the same table (whether the other rows precede or follow them) are effectively padded on the right with empty `mtd` elements so that the number of columns in each row equals the maximum number of columns in any row of the table. Note that the use of `mtd` elements with non-default values of the `rowspan` or `columnspan` attributes may affect the number of `mtd` elements that should be given in subsequent `mtr` elements to cover a given number of columns. Note also that the label in an `mtlabeledtr` element is not considered a column in the table.

3.5.1.2 Attributes

Name	values	default
align	(top | bottom | center | baseline | axis) [rownumber]	axis
rowalign	(top | bottom | center | baseline | axis) +	baseline
columnalign	(left | center | right) +	center
groupalign	group-alignment-list-list	left
alignmentscope	(true | false) +	true
columnwidth	(auto | number h-unit | namedspace | fit) +	auto
width	auto | number h-unit	auto
rowspacing	(number v-unit) +	1.0ex
columnspacing	(number h-unit | namedspace) +	0.8em
rowlines	(none | solid | dashed) +	none
columnlines	(none | solid | dashed) +	none
frame	none | solid | dashed	none
framespacing	(number h-unit | namedspace) (number v-unit | namedspace)	0.4em 0.5ex
equalrows	true | false	true
equalcolumns	true | false	true
displaystyle	true | false	false
side	left | right | leftoverlap | rightoverlap	right
minlabelspacing	number h-unit	0.8em

Note that the default value for each of `rowlines`, `columnlines` and `frame` is the literal string 'none', meaning that the default is to render no lines, rather than that there is no default.

As described in Section 2.3.4, the notation `(x | y)+` means one or more occurrences of either `x` or `y`, separated by whitespace. For example, possible values for `columnalign` are `left`, `left left`, and `left right center center`. If there are more entries than are necessary (e.g. more entries than columns for `columnalign`), then only the first entries will be used. If there are fewer entries, then the last entry is repeated as often as necessary. For example, if `columnalign`="right center" and the table has three columns, the first column will be right aligned and the second and third columns will be centered. The label in a `mlabeledtr` is not considered as a column in the table and the attribute values that apply to columns do not apply to labels

The `align` attribute specifies where to align the table with respect to its environment. 'axis' means to align the center of the table on the environment's axis. (The axis of an equation is an alignment line used by typesetters. It is the line on which a minus sign typically lies. The center of the table is the midpoint of the table's vertical extent.) 'center' and 'baseline' both mean to align the center of the table on the environment's baseline. 'top' or 'bottom' aligns the top or bottom of the table on the environment's baseline.

If the `align` attribute value ends with a `rownumber` between 1 and n (for a table with n rows), the specified row is aligned in the way described above, rather than the table as a whole; the top (first) row is numbered 1, and the bottom (last) row is numbered n. The same is true if the row number is negative, between -1 and $-n$, except that the bottom row is referred to as -1 and the top row as $-n$. Other values of `rownumber` are illegal.

The `rowalign` attribute specifies how the entries in each row should be aligned. For example, 'top' means that the tops of each entry in each row should be aligned with the tops of the other entries in that row. The `columnalign` attribute specifies how the entries in each column should be aligned.

The `groupalign` and `alignmentscope` attributes are described with the alignment elements, `maligngroup` and `malignmark`, in Section 3.5.5.

The `columnwidth` attribute specifies how wide a column should be. The `auto` value means that the column should be as wide as needed, which is the default. If an explicit value is given, then the column is exactly that wide and the contents of that column are made to fit in that width. The contents are linewrapped or clipped at the discretion of the renderer. If `fit` is given as a value, the remaining page width after subtracting the widths for columns specified as `auto` and/or specific widths is divided equally among the `fit` columns and this value is used for the column width. If insufficient room remains to hold the contents of the `fit` columns, renderers may linewrap or clip the contents of the `fit` columns. When the `columnwidth` is specified as a percentage, the value is relative to the width of the table. That is, a renderer should try to adjust the width of the column so that it covers the specified percentage of the entire table width.

The `width` attribute specifies the desired width of the entire table and is intended for visual user agents. When the value is a percentage value, the value is relative to the horizontal space a MathML renderer has available for the table element. When the value is `auto`, the MathML renderer should calculate the table width from its contents using whatever layout algorithm it chooses.

MathML 2.0 does not specify a table layout algorithm. In particular, it is the responsibility of a MathML renderer to resolve conflicts between the `width` attribute and other constraints on the width of a table, such as explicit values for `columnwidth` attributes, and minimum sizes for table cell contents. For a discussion of table layout algorithms, see Cascading Style Sheets, level 2.

The `rowspacing` and `columnspacing` attributes specify how much space should be added between each row and column. However, spacing before the first row and after the last row (i.e. at the top and bottom of the table) is given by the second number in the value of the `framespacing` attribute, and spacing before the first column and after the last column (i.e. on the left and on the right of the table) is given by the first number in the value of the `framespacing` attribute.

In those attributes' syntaxes, *h-unit* or *v-unit* represents a unit of horizontal or vertical length, respectively (see Section 2.3.4.2). The units shown in the attributes' default values (`em` or `ex`) are typically used.

The `rowlines` and `columnlines` attributes specify whether and what kind of lines should be added between each row and column. Lines before the first row or column and after the last row or column are given using the `frame` attribute.

If a frame is desired around the table, the `frame` attribute is used. If the attribute value is not 'none', then `framespacing` is used to add spacing between the lines of the frame and the first and last rows and columns of the table. If `frame=`"none", then the `framespacing` attribute is ignored. The `frame` and `framespacing` attributes are not part of the `rowlines`/`columnlines`, `rowspacing`/`columnspacing` options because having them be so would often require that `rowlines` and `columnlines` would need to be fully specified instead of just giving a single value. For example, if a table had five columns and we wanted lines between the columns, but no frame, then we would have to write `columnlines="none solid solid solid solid none"`. By separating the frame from the internal lines, we only need to write `columnlines="solid"`.

The `equalrows` attribute forces the rows all to be the same total height when set to `true`. The `equalcolumns` attribute forces the columns all to be the same width when set to `true`.

The `displaystyle` attribute specifies the value of `displaystyle` (described under `mstyle` in Section 3.3.4) within each cell (`mtd` element) of the table. Setting `displaystyle=true` can be useful for tables whose elements are whole mathematical expressions; the default value of `false` is appropriate when the table is part of an expression, for example, when it represents a matrix. In either case, `scriptlevel` (Section 3.3.4) is not changed for the table cells.

The `side` attribute specifies what side of a table a label for a table row should should be placed. This attribute is intended to be used for labeled expressions. If `left` or `right` is specified, the label is placed on the left or right side of the table row respectively. The other two attribute values are variations on `left` and `right`: if the labeled row fits within the width allowed for the table without the label, but does not fit within the width if the label is included, then the label overlaps the row and is displayed above the row if `rowalign` for that row is `top`; otherwise the label is displayed below the row.

If there are multiple labels in a table, the alignment of the labels within the virtual column that they form is left-aligned for labels on the left side of the table, and right-aligned for labels on the right side of the table. The alignment can be overridden by specifying `columnalignment` for a `mlabeltr` element.

The `minlabelspacing` attribute specifies the minimum space allowed between a label and the adjacent entry in the row.

3.5.1.3 Examples

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

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

This might be rendered as:

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

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

3.5.2 Row in Table or Matrix (`mtr`)

3.5.2.1 Description

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

As described in Section 3.5.1, `mtr` elements are effectively padded on the right with `mtd` elements when they are shorter than other rows in a table.

3.5.2.2 Attributes

Name	values	default
rowalign	top | bottom | center | baseline | axis	inherited
columnalign	(left | center | right) +	inherited
groupalign	group-alignment-list-list	inherited

The `rowalign` and `columnalign` attributes allow a specific row to override the alignment specified by the same attributes in the surrounding `mtable` element.

As with `mtable`, if there are more entries than necessary in the value of `columnalign` (i.e. more entries than columns in the row), then the extra entries will be ignored. If there are fewer entries than columns, then the last entry will be repeated as many times as needed.

The `groupalign` attribute is described with the alignment elements, `maligngroup` and `malignmark`, in Section 3.5.5.

3.5.3 Labeled Row in Table or Matrix (`mlabeledtr`)

3.5.3.1 Description

An `mlabeledtr` element represents one row in a table that has a label on either the left or right side, as determined by the `side` attribute. The label is the first child of `mlabeledtr`. The rest of the children represent the contents of the row and are identical to those used for `mtr`; all of the children except the first must be `mtd` elements.

An `mlabeledtr` element is only allowed as a direct sub-expression of an `mtable` element. Each argument of `mlabeledtr` except for the first argument (the label) is placed in a different column of the table, starting at the leftmost column.

Note that the label element is not considered to be a cell in the table row. In particular, the label element is not taken into consideration in the table layout for purposes of width and alignment calculations. For example, in the case of an mlabeledtr with a label and a single centered mtd child, the child is first centered in the enclosing mtable, and then the label is placed. Specifically, the child is *not* centered in the space that remains in the table after placing the label.

While MathML 2.0 does not specify an algorithm for placing labels, implementors of visual renderers may find the following formatting model useful. To place a label, an implementor might think in terms of creating a larger table, with an extra column on both ends. The columnwidth attributes of both these border columns would be set to fit so that they expand to fill whatever space remains after the inner columns have be laid out. Finally, depending on the values of side and minlabelspacing, the label is placed in whatever border column is appropriate, possibly shifted down if necessary.

3.5.3.2 Attributes

The attributes for mlabeledtr are the same as for mtr. Unlike the attributes for the mtable element, attributes of mlabeledtr that apply to column elements also apply to the label. For example, in a one column table,

```
<mlabeledtr rowalign='center baseline'>
```

means that the label is vertically centered on the row, and that the actual entry is baseline aligned.

3.5.3.3 Equation Numbering

One of the important uses of mlabeledtr is for numbered equations. In a mlabeledtr, the label represents the equation number and the elements in the row are the equation being numbered. The side and minlabelspacing attributes of of mtable determine the placement of the equation number.

In larger documents with many numbered equations, automatic numbering becomes important. While automatic equation numbering and automatically resolving references to equation numbers is outside the scope of MathML, these problems can be addressed by the use of style sheets or other means. The mlabeledtr construction provides support for both of these functions in a way that is intended to facilitate XSL processing. The mlabeledtr element can be used to indicate the presence of a numbered equation, and the first child can be changed to the current equation number, along with incrementing the global equation number. For cross references, an id on either the mlabeledtr element or on the first element itself could be used as a target of any link.

```
<mtable>
  <mlabeledtr id='e-is-m-c-square'>
    <mtext> (2.1) </mtext>
    <mtd>
     <mrow>
       <mi>E</mi>
       <mo>=</mo>
       <mrow>
        <mi>m</mi>
        <mo>&it;</mo>
        <msup>
         <mi>c</mi>
         <mn>2</mn>
        </msup>
       </mrow>
     </mrow>
    </mtd>
  </mlabeledtr>
</mtable>
```

This should be rendered as:

$$E = mc\hat{}2 \tag{2.1}$$

3.5.4 Entry in Table or Matrix (mtd)

3.5.4.1 Description

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

The mtd element accepts any number of arguments; if this number is not 1, its contents are treated as a single 'inferred mrow' formed from all its arguments, as described in Section 3.1.3.

3.5.4.2 Attributes

Name	values	default
rowspan	number	1
columnspan	number	1
rowalign	top | bottom | center | baseline | axis	inherited
columnalign	left | center | right	inherited
groupalign	group-alignment-list	inherited

The rowspan and columnspan attributes allow a specific matrix element to be treated as if it occupied the number of rows or columns specified. The interpretation of how this larger element affects specifying subsequent rows and columns is meant to correspond with the similar attributes for HTML 4.0 tables.

The rowspan and columnspan attributes can be used around an mtd element that represents the label in a mlabeledtr element. Also, the label of a mlabeledtr element is not considered to be part of a previous rowspan and columnspan.

The rowalign and columnalign attributes allow a specific matrix element to override the alignment specified by a surrounding mtable or mtr element.

The groupalign attribute is described with the alignment elements, maligngroup and malignmark, in Section 3.5.5.

3.5.5 Alignment Markers

3.5.5.1 Description

These are space-like elements (see Section 3.2.6) that can be used to vertically align specified points within a column of MathML expressions, by the automatic insertion of the necessary amount of horizontal space between specified sub-expressions.

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

```
8.44x + 55  y =  0
3.1 x -  0.7y = -1.1
```

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

```
8.44x + 55y = 0
3.1x - 0.7y = -1.1
```

(For audio renderers, it is suggested that the alignment elements produce the analogous behavior of altering the rhythm of pronunciation so that it is the same for several sub-expressions in a column, by the insertion of the appropriate time delays in place of the extra horizontal spacing described here.)

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

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

The reason `mtable` columns are used as alignment scopes is that they are the only general way in MathML to arrange expressions into vertical columns. Future versions of MathML may provide an `malignscope` element that allows an alignment scope to be created around any MathML element, but even then, table columns would still sometimes need to act as alignment scopes, and since they are not elements themselves, but rather are made from corresponding parts of the content of several `mtr` elements, they could not individually be the content of an alignment scope element.

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

3.5.5.2 Specifying alignment groups

To cause alignment, it is necessary to specify, within each expression to be aligned, the points to be aligned with corresponding points in other expressions, and the beginning of each *alignment group* of sub-expressions that can be horizontally shifted as a unit to effect the alignment. Each alignment group must contain one alignment point. It is also necessary to specify which expressions in the column have no alignment groups at all, but are affected only by the ordinary column alignment for that column of the table, i.e. by the `columnalign` attribute, described elsewhere.

The alignment groups start at the locations of invisible `maligngroup` elements, which are rendered with zero width when they occur outside of an alignment scope, but within an alignment scope are rendered with just enough horizontal space to cause the desired alignment of the alignment group that follows them. A simple algorithm by which a MathML application can achieve this is given later. In the example above, each equation would have one `maligngroup` element before each coefficient, variable, and operator on the left-hand side, one before the = sign, and one before the constant on the right-hand side.

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

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

The nested expression structure formed by `mrow`s and other layout schemata should reflect the mathematical structure of the expression, not the alignment-group structure, to make possible optimal renderings and better automatic interpretations; see the discussion of proper grouping in section Section 3.3.1. Insertion of alignment elements (or other space-like elements) should not alter the correspondence between the structure of a MathML expression and the structure of the mathematical expression it represents.

Although alignment groups need to coincide with the nested expression structure of layout schemata, there are nonetheless restrictions on where an `maligngroup` element is allowed within a table cell. The `maligngroup` element may only be contained within elements of the following types (which are themselves contained in the table cell):

- an `mrow` element, including an inferred `mrow` such as the one formed by a multi-argument `mtd` element;
- an `mstyle` element;
- an `mphantom` element;
- an `mfenced` element;
- an `maction` element, though only its selected sub-expression is checked;
- a `semantics` element.

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

Note that some positions for an `maligngroup` element, although legal, are not useful, such as for an `maligngroup` element to be an argument of an `mfenced` element. When inserting an `maligngroup` element before a given element in pre-existing MathML, it will often be necessary, and always acceptable, to form a new `mrow` element to contain just the `maligngroup` element and the element it is inserted before. In general, this will be necessary except when the `maligngroup` element is inserted directly into an `mrow` or into an element that can form an inferred `mrow` from its contents. See the warning about the legal grouping of 'space-like elements' in Section 3.2.6.

For the table cells that are divided into alignment groups, every element in their content must be part of exactly one alignment group, except the elements from the above list that contain `maligngroup` elements inside them, and the `maligngroup` elements themselves. This means that, within any table cell containing alignment groups, the first complete element must be an `maligngroup` element, though this may be preceded by the begin tags of other elements.

This requirement removes a potential confusion about how to align elements before the first `maligngroup` element, and makes it easy to identify table cells that are left out of their column's alignment process entirely.

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

3.5.5.3 Table cells that are not divided into alignment groups

Expressions in a column that are to have no alignment groups should contain no `maligngroup` elements. Expressions with no alignment groups are aligned using only the `columnalign` attribute that applies to the table column as a whole, and are not affected by the `groupalign` attribute described below. If such an expression is wider than the column width needed for the table cells containing alignment groups, all the table cells containing alignment groups will be shifted as a unit within the column as described by the `columnalign` attribute for that column. For example, a column heading with no internal alignment could be added to the column of two equations given above by preceding them with another table row containing an `mtext` element for the heading, and using the default `columnalign`="center" for the table, to produce:

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

or, with a shorter heading,

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

3.5.5.4 Specifying alignment points using `malignmark`

Each alignment group's alignment point can either be specified by an `malignmark` element anywhere within the alignment group (except within another alignment scope wholly contained inside it), or it is determined automatically from the `groupalign` attribute. The `groupalign` attribute can be specified on the group's preceding `maligngroup` element or on its surrounding `mtd`, `mtr`, or `mtable` elements. In typical cases, using the `groupalign` attribute is sufficient to describe the desired alignment points, so no `malignmark` elements need to be provided.

The `malignmark` element indicates that the alignment point should occur on the right edge of the preceding element, or the left edge of the following element or character, depending on the `edge` attribute of `malignmark`. Note that it may be necessary to introduce an `mrow` to group an `malignmark` element with a neighboring element, in order not to alter the argument count of the containing element. (See the warning about the legal grouping of 'space-like elements' in Section 3.2.6).

When an `malignmark` element is provided within an alignment group, it can occur in an arbitrarily deeply nested element within the group, as long as it is not within a nested alignment scope. It is not subject to the same restrictions on location as `maligngroup` elements. However, its immediate surroundings need to be such that the element to its immediate right or left (depending on its `edge` attribute) can be unambiguously identified. If no such element is present, renderers should behave as if a zero-width element had been inserted there.

For the purposes of alignment, an element X is considered to be to the immediate left of an element Y, and Y to the immediate right of X, whenever X and Y are successive arguments of one (possibly inferred) `mrow` element, with X coming before Y. In the case of `mfenced` elements, MathML applications should evaluate this relation as if the `mfenced` element had been replaced by the equivalent expanded form involving `mrow`. Similarly, an `maction` element should be treated as if it were replaced by its currently selected sub-expression. In all other cases, no relation of 'to the immediate left or right' is defined for two elements X and Y. However, in the case of content elements interspersed in presentation markup, MathML applications should attempt to evaluate this relation in a sensible way. For example, if a renderer maintains an internal presentation structure for rendering content elements, the relation could be evaluated with respect to that. (See Chapter 4 and Chapter 5 for further details about mixing presentation and content markup.)

Unlike all other elements in MathML, `malignmark` elements are allowed to occur within the content of token elements, such as `mn`, `mi`, or `mtext`. When this occurs, the character immediately before or after the `malignmark` element will carry the alignment point; in all other cases, the element to its immediate left or right will carry the alignment point. The rationale for this is that it is sometimes desirable to align on the edges of specific characters within multi-character token elements.

If there is more than one `malignmark` element in an alignment group, all but the first one will be ignored. MathML applications may wish to provide a mode in which they will warn about this situation, but it is not an error, and should trigger no warnings by default. (Rationale: it would be inconvenient to have to remove all unnecessary `malignmark` elements from automatically generated data, in certain cases, such as when they are used to specify alignment on 'decimal points' other than the '.' character.)

3.5.5.5 Attributes

Name	values	default
edge	left | right	left

`malignmark` has one attribute, `edge`, which specifies whether the alignment point will be found on the left or right edge of some element or character. The precise location meant by 'left edge' or 'right edge' is discussed below. If `edge`="right", the alignment point is the right edge of the element or character to the immediate left of the `malignmark` element. If `edge`="left", the alignment point is the left edge of the element or character to the immediate right of the `malignmark` element. Note that the attribute refers to the choice of edge rather than to the direction in which to look for the element whose edge will be used.

For `malignmark` elements that occur within the content of MathML token elements, the preceding or following character in the token element's content is used; if there is no such character, a zero-width character is effectively inserted for the purpose of carrying the alignment point on its edge. For all other `malignmark` elements, the preceding or following element is used; if there is no such element, a zero-width element is effectively inserted to carry the alignment point.

The precise definition of the 'left edge' or 'right edge' of a character or glyph (e.g. whether it should coincide with an edge of the character's bounding box) is not specified by MathML, but is at the discretion of the renderer; the renderer is allowed to let the edge position depend on the character's context as well as on the character itself.

For proper alignment of columns of numbers (using `groupalign` values of `left`, `right`, or `decimalpoint`), it is likely to be desirable for the effective width (i.e. the distance between the left and right edges) of decimal digits to be constant, even if their bounding box widths are not constant (e.g. if '1' is narrower than other digits). For other characters, such as letters and operators, it may be desirable for the aligned edges to coincide with the bounding box.

The 'left edge' of a MathML element or alignment group refers to the left edge of the leftmost glyph drawn to render the element or group, except that explicit space represented by `mspace` or `mtext` elements should also count as 'glyphs' in this context, as should glyphs that would be drawn if not for `mphantom` elements around them. The 'right edge' of an element or alignment group is defined similarly.

3.5.5.6 Attributes

Name	values	default
groupalign	left | center | right | decimalpoint	inherited

`maligngroup` has one attribute, `groupalign`, which is used to determine the position of its group's alignment point when no `malignmark` element is present. The following discussion assumes that no `malignmark` element is found within a group.

In the example given at the beginning of this section, there is one column of 2 table cells, with 7 alignment groups in each table cell; thus there are 7 columns of alignment groups, with 2 groups, one above the other, in each column. These columns of alignment groups should be given the 7 `groupalign` values ‘decimalpoint left left decimalpoint left left decimalpoint’, in that order. How to specify this list of values for a table cell or table column as a whole, using attributes on elements surrounding the `maligngroup` element is described later.

If `groupalign` is ‘left’, ‘right’, or ‘center’, the alignment point is defined to be at the group’s left edge, at its right edge, or halfway between these edges, respectively. The meanings of ‘left edge’ and ‘right edge’ are as discussed above in relation to `malignmark`.

If `groupalign` is ‘decimalpoint’, the alignment point is the right edge of the last character before the decimal point. The decimal point is the first ‘.’ character (ASCII 0x2e) in the first `mn` element found along the alignment group’s baseline. More precisely, the alignment group is scanned recursively, depth-first, for the first `mn` element, descending into all arguments of each element of the types `mrow` (including inferred `mrows`), `mstyle`, `mpadded`, `mphantom`, `mfenced`, or `msqrt`, descending into only the first argument of each ‘scripting’ element (`msub`, `msup`, `msubsup`, `munder`, `mover`, `munderover`, `mmultiscripts`) or of each `mroot` or `semantics` element, descending into only the selected sub-expression of each `maction` element, and skipping the content of all other elements. The first `mn` so found always contains the alignment point, which is the right edge of the last character before the first decimal point in the content of the `mn` element. If there is no decimal point in the `mn` element, the alignment point is the right edge of the last character in the content. If the decimal point is the first character of the `mn` element’s content, the right edge of a zero-width character inserted before the decimal point is used. If no `mn` element is found, the right edge of the entire alignment group is used (as for `groupalign`="right").

In order to permit alignment on decimal points in `cn` elements, a MathML application can convert a content expression into a presentation expression that renders the same way before searching for decimal points as described above.

If characters other than ‘.’ should be used as ‘decimal points’ for alignment, they should be preceded by `malignmark` elements within the `mn` token’s content itself.

For any of the `groupalign` values, if an explicit `malignmark` element is present anywhere within the group, the position it specifies (described earlier) overrides the automatic determination of alignment point from the `groupalign` value.

3.5.5.7 Inheritance of `groupalign` values

It is not usually necessary to put a `groupalign` attribute on every `maligngroup` element. Since this attribute is usually the same for every group in a column of alignment groups to be aligned, it can be inherited from an attribute on the `mtable` that was used to set up the alignment scope as a whole, or from the `mtr` or `mtd` elements surrounding the alignment group. It is inherited via an ‘inheritance path’ that proceeds from `mtable` through successively contained `mtr`, `mtd`, and `maligngroup` elements. There is exactly one element of each of these kinds in this path from an `mtable` to any alignment group inside it. In general, the value of `groupalign` will be inherited by any given alignment group from the innermost element that surrounds the alignment group and provides an explicit setting for this attribute.

Note, however, that each `mtd` element needs, in general, a list of `groupalign` values, one for each `maligngroup` element inside it, rather than just a single value. Furthermore, an `mtr` or `mtable` element needs, in general, a list of lists of `groupalign` values, since it spans multiple `mtable` columns, each potentially acting as an alignment scope. Such lists of group-alignment values are specified using the following syntax rules:

```
group-alignment            := left | right | center | decimalpoint
group-alignment-list       := group-alignment +
group-alignment-list-list := ( ’{’ group-alignment-list ’}’ ) +
```

As described in Section 2.3.4, `|` separates alternatives; `+` represents optional repetition (i.e. 1 or more copies of what precedes it), with extra values ignored and the last value repeated if necessary to cover additional table columns or alignment group columns; `’’ and ’’` represent literal braces; and `(` and `)` are used for grouping, but do not literally appear in the attribute value.

The permissible values of the `groupalign` attribute of the elements that have this attribute are specified using the above syntax definitions as follows:

Element type	groupalign attribute syntax	default value
`mtable`	group-alignment-list-list	left
`mtr`	group-alignment-list-list	inherited from `mtable` attribute
`mtd`	group-alignment-list	inherited from within `mtr` attribute
`maligngroup`	group-alignment	inherited from within `mtd` attribute

In the example near the beginning of this section, the group alignment values could be specified on every `mtd` element using `groupalign` = ‘decimalpoint left left decimalpoint left left decimalpoint’, or on every `mtr` element using `groupalign` = ‘decimalpoint left left decimalpoint left left decimalpoint’, or (most conveniently) on the `mtable` as a whole using `groupalign` = ‘decimalpoint left left decimalpoint left left decimalpoint’, which provides a single braced list of group-alignment values for the single column of expressions to be aligned.

3.5.5.8 MathML representation of an alignment example

The above rules are sufficient to explain the MathML representation of the example given near the start of this section. To repeat the example, the desired rendering is:

```
8.44x + 55  y =  0
3.1 x -  0.7y = -1.1
```

One way to represent that in MathML is:

```
<mtable groupalign="decimalpoint left left decimalpoint left left decimalpoint">
  <mtd>
    <mrow>
      <mrow>
        <malignmark/>
        <mn> 8.44 </mn>
        <mo> &InvisibleTimes; </mo>
        <maligngroup/>
        <mi> x </mi>
      </mrow>
      <maligngroup/>
      <mo> + </mo>
      <mrow>
        <maligngroup/>
        <mn> 55 </mn>
        <mo> &InvisibleTimes; </mo>
        <maligngroup/>
        <mi> y </mi>
      </mrow>
    </mrow>
    <maligngroup/>
    <mo> = </mo>
    <maligngroup/>
    <mn> 0 </mn>
  </mtd>
  <mtd>
    <mrow>
      <mrow>
        <maligngroup/>
        <mn> 3.1 </mn>
        <mo> &InvisibleTimes; </mo>
        <maligngroup/>
        <mi> x </mi>
      </mrow>
```

```
      <maligngroup/>
      <mo> - </mo>
      <mrow>
        <maligngroup/>
        <mn> 0.7 </mn>
        <mo> &InvisibleTimes; </mo>
        <maligngroup/>
        <mi> y </mi>
      </mrow>
    </mrow>
    <maligngroup/>
    <mo> = </mo>
    <maligngroup/>
    <mrow>
      <mo> - </mo>
      <mn> 1.1 </mn>
    </mrow>
  </mtd>
</mtable>
```

3.5.5.9 Further details of alignment elements

The alignment elements `malignroup` and `malignmark` can occur outside of alignment scopes, where they are ignored. The rationale behind this is that in situations in which MathML is generated, or copied from another document, without knowing whether it will be placed inside an alignment scope, it would be inconvenient for this to be an error.

An `mtable` element can be given the attribute `alignmentscope=false` to cause its columns not to act as alignment scopes. In general, this attribute has the syntax `(true | false) +`; if its value is a list of boolean values, each boolean value applies to one column, with the last value repeated if necessary to cover additional columns, or with extra values ignored. Columns that are not alignment scopes are part of the alignment scope surrounding the `mtable` element, if there is one. Use of `alignmentscope=false` allows nested tables to contain `malignmark` elements for aligning the inner table in the surrounding alignment scope.

As discussed above, processing of alignment for content elements is not well-defined, since MathML does not specify how content elements should be rendered. However, many MathML applications are likely to find it convenient to internally convert content elements to presentation elements that render the same way. Thus, as a general rule, even if a renderer does not perform such conversions internally, it is recommended that the alignment elements should be processed as if it did perform them.

A particularly important case for renderers to handle gracefully is the interaction of alignment elements with the `matrix` content element, since this element may or may not be internally converted to an expression containing an `mtable` element for rendering. To partially resolve this ambiguity, it is suggested, but not required, that if the `matrix` element is converted to an expression involving an `mtable` element, that the `mtable` element be given the attribute `alignmentscope=false`, which will make the interaction of the `matrix` element with the alignment elements no different than that of a generic presentation element (in particular, it will allow it to contain `malignmark` elements that operate within the alignment scopes created by the columns of an `mtable` that contains the `matrix` element in one of its table cells).

The effect of alignment elements within table cells that have non-default values of the `columnspan` or `rowspan` attributes is not specified, except that such use of alignment elements is not an error. Future versions of MathML may specify the behavior of alignment elements in such table cells.

The effect of possible linebreaking of an `mtable` element on the alignment elements is not specified.

3.5.5.10 A simple alignment algorithm

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

First, a rendering is computed for the contents of each table cell in the column, using zero width for all `maligngroup` and `malignmark` elements. The final rendering will be identical except for horizontal shifts applied to each alignment group and/or table cell. The positions of alignment points specified by any `malignmark` elements are noted, and the remaining alignment points are determined using `groupalign` values.

For each alignment group, the horizontal positions of the left edge, alignment point, and right edge are noted, allowing the width of the group on each side of the alignment point (left and right) to be determined. The sum of these two 'side-widths', i.e. the sum of the widths to the left and right of the alignment point, will equal the width of the alignment group.

Second, each column of alignment groups, from left to right, is scanned. The ith scan covers the ith alignment group in each table cell containing any alignment groups. Table cells with no alignment groups, or with fewer than i alignment groups, are ignored. Each scan computes two maximums over the alignment groups scanned: the maximum width to the left of the alignment point, and the maximum width to the right of the alignment point, of any alignment group scanned.

The sum of all the maximum widths computed (two for each column of alignment groups) gives one total width, which will be the width of each table cell containing alignment groups. Call the maximum number of alignment groups in one cell n; each such cell's width is divided into $2n$ adjacent sections, called L(i) and R(i) for i from 1 to n, using the $2n$ maximum side-widths computed above; for each i, the width of all sections called L(i) is the maximum width of any cell's ith alignment group to the left of its alignment point, and the width of all sections called R(i) is the maximum width of any cell's ith alignment group to the right of its alignment point.

The alignment groups are then positioned in the unique way that places the part of each ith group to the left of its alignment point in a section called L(i), and places the part of each ith group to the right of its alignment point in a section called R(i). This results in the alignment point of each ith group being on the boundary between adjacent sections L(i) and R(i), so that all alignment points of ith groups have the same horizontal position.

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

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

3.6 Enlivening Expressions

3.6.1 Bind Action to Sub-Expression (`maction`)

There are many ways in which it might be desirable to make mathematical content active. Adding a link to a MathML sub-expressions is one basic kind of interactivity Section 7.1.4. However, many other kinds of interactivity cannot be easily accommodated by generic linking mechanisms. For example, in lengthy mathematical expressions, the ability to 'fold' expressions might be provided, i.e. a renderer might allow a reader to toggle between an ellipsis and a much longer expression that it represents.

To provide a mechanism for binding actions to expressions, MathML provides the `maction` element. This element accepts any number of sub-expressions as arguments, and the following attributes:

Name	values	default
actiontype	(described below)	(required attribute, no default value)
selection	positive-integer	1

By default, MathML applications that do not recognize the specified `actiontype` should render the selected sub-expression as defined below. If no selected sub-expression exists, it is a MathML error; the appropriate rendering in that case is as described in Section 7.2.2 on the treatment of MathML errors.

Since a MathML-compliant application is not required to recognize any particular `actiontype`s, an application can be fully MathML compliant just by implementing the above-described default behavior.

The `selection` attribute is provided for those `actiontype`s that permit someone viewing a document to select one of several sub-expressions for viewing. Its value should be a positive integer that indicates one of the sub-expressions of the `maction` element, numbered from 1 to the number of children of the element. When this is the case, the sub-expression so indicated is defined to be the 'selected sub-expression' of the `maction` element; otherwise the 'selected sub-expression' does not exist, which is an error. When the `selection` attribute is not specified (including for actiontypes for which it makes no sense), its default value is 1, so the selected sub-expression will be the first sub-expression.

Furthermore, as described in Chapter 7, if a MathML application responds to a user command to copy a MathML sub-expression to the environment's 'clipboard', any `maction` elements present in what is copied should be given selection attributes that correspond to their selection state in the MathML rendering at the time of the copy command.

A suggested list of `actiontype`s and their associated actions is given below. Keep in mind, however, that this list is mainly for illustration, and recognized values and behaviors will vary from application to application.

<maction actiontype="toggle" selection="positive-integer" > (first expression) (second expression)... </maction>

For this action type, a renderer would alternately display the given expressions, cycling through them when a reader clicked on the active expression, starting with the selected expression and updating the `selection` attribute value as described above. Typical uses would be for exercises in education, ellipses in long computer algebra output, or to illustrate alternate notations. Note that the expressions may be of significantly different size, so that size negotiation with the browser may be desirable. If size negotiation is not available, scrolling, elision, panning, or some other method may be necessary to allow full viewing.

<maction actiontype="statusline"> (expression) (message) </maction>

In this case, the renderer would display the expression in context on the screen. When a reader clicked on the expression or moved the mouse over it, the renderer would send a rendering of the message to the browser statusline. Since most browsers in the foreseeable future are likely to be limited to displaying text on their statusline, authors would presumably use plain text in an `mtext` element for the message in most circumstances. For non-`mtext` messages, renderers might provide a natural language translation of the markup, but this is not required.

<maction actiontype="tooltip"> (expression) (message) </maction>

Here the renderer would also display the expression in context on the screen. When the mouse pauses over the expression for a long enough delay time, the renderer displays a rendering of the message in a pop-up 'tooltip' box near the expression. These message boxes are also sometimes called 'balloon help' boxes. Presumably authors would use plain text in an `mtext` element for the message in most circumstances. For non-`mtext` messages, renderers may provide a natural language translation of the markup if full MathML rendering is not practical, but this is not required.

<maction actiontype="highlight" my:color="#ff0000"> expression </maction> <maction actiontype="highlight" my:background="#ff0000"> expression </maction>

In this case, a renderer might highlight the enclosed expression on a 'mouse-over' event. In the example given above, non-standard attributes from another namespace are being used to pass additional information to renderers that support them, without violating the MathML DTD (see Section 7.2.3). The `my:color` attribute changes the color of the characters in the presentation, while the `my:background` attribute changes the color of the background behind the characters.

<maction actiontype="menu" selection="1" > (menu item 1) (menu item 2) ... </maction>

This action type instructs a renderer to provide a pop up menu. This allows a one-to-many linking capability. Note that the menu items may be other <maction actiontype="menu">...</maction> expressions, thereby allowing nested menus.

Chapter 4

Content Markup

4.1 Introduction

4.1.1 The Intent of Content Markup

As has been noted in the introductory section of this recommendation, mathematics can be distinguished by its use of a (relatively) formal language, mathematical notation. However, mathematics and its presentation should not be viewed as one and the same thing. Mathematical sums or products exist and are meaningful to many applications completely without regard to how they are rendered aurally or visually. The intent of the content markup in the Mathematical Markup Language is to provide an explicit encoding of the *underlying mathematical structure* of an expression, rather than any particular rendering for the expression.

There are many reasons for providing a specific encoding for content. Even a disciplined and systematic use of presentation tags cannot properly capture this semantic information. This is because without additional information it is impossible to decide if a particular presentation was chosen deliberately to encode the mathematical structure or simply to achieve a particular visual or aural effect. Furthermore, an author using the same encoding to deal with both the presentation and mathematical structure might find a particular presentation encoding unavailable simply because convention had reserved it for a different semantic meaning.

The difficulties stem from the fact that there are many to one mappings from presentation to semantics and vice versa. For example the mathematical construct 'H multiplied by e' is often encoded using an explicit operator as in $H \times e$. In different presentational contexts, the multiplication operator might be invisible '$H\ e$', or rendered as the spoken word 'times'. Generally, many different presentations are possible depending on the context and style preferences of the author or reader. Thus, given '$H\ e$' out of context it may be impossible to decide if this is the name of a chemical or a mathematical product of two variables H and e.

Mathematical presentation also changes with culture and time: some expressions in combinatorial mathematics today have one meaning to an Russian mathematician, and quite another to a French mathematician; see Section 5.4.1 for an example. Notations may lose currency, for example the use of musical sharp and flat symbols to denote maxima and minima [Chaundy1954]. A notation in use in 1644 for the multiplication mentioned above was ■ $H\ e$ [Cajori1928].

When we encode the underlying mathematical structure explicitly, without regard to how it is presented aurally or visually, we are able to interchange information more precisely with those systems that are able to manipulate the mathematics. In the trivial example above, such a system could substitute values for the variables H and e and evaluate the result. Further interesting application areas include interactive textbooks and other teaching aids.

4.1.2 The Scope of Content Markup

The semantics of general mathematical notation is not a matter of consensus. It would be an enormous job to systematically codify most of mathematics - a task that can never be complete. Instead, MathML makes explicit a relatively small number of commonplace mathematical constructs, chosen carefully to be sufficient in a large number of applications. In addition, it provides a mechanism for associating semantics with new notational constructs. In this way, mathematical concepts that are not in the base collection of elements can still be encoded (Section 4.2.6).

The base set of content elements are chosen to be adequate for simple coding of most of the formulas used from kindergarten to the end of high school in the United States, and probably beyond through the first two years of college, that is up to A-Level or Baccalaureate level in Europe. Subject areas covered to some extent in MathML are:

- arithmetic, algebra, logic and relations
- calculus and vector calculus
- set theory
- sequences and series
- elementary classical functions
- statistics
- linear algebra

It is not claimed, or even suggested, that the proposed set of elements is complete for these areas, but the provision for author extensibility greatly alleviates any problem omissions from this finite list might cause.

4.1.3 Basic Concepts of Content Markup

The design of the MathML content elements are driven by the following principles:

- The expression tree structure of a mathematical expression should be directly encoded by the MathML content elements.
- The encoding of an expression tree should be explicit, and not dependent on the special parsing of `PCDATA` or on additional processing such as operator precedence parsing.
- The basic set of mathematical content constructs that are provided should have default mathematical semantics.
- There should be a mechanism for associating specific mathematical semantics with the constructs.

The primary goal of the content encoding is to establish explicit connections between mathematical structures and their mathematical meanings. The content elements correspond directly to parts of the underlying mathematical expression tree. Each structure has an associated default semantics and there is a mechanism for associating new mathematical definitions with new constructs.

Significant advantages to the introduction of content-specific tags include:

- Usage of presentation elements is less constrained. When mathematical semantics are inferred from presentation markup, processing agents must either be quite sophisticated, or they run the risk of inferring incomplete or incorrect semantics when irregular constructions are used to achieve a particular aural or visual effect.
- It is immediately clear which kind of information is being encoded simply by the kind of elements that are used.
- Combinations of semantic and presentation elements can be used to convey both the appearance and its mathematical meaning much more effectively than simply trying to infer one from the other.

Expressions described in terms of content elements must still be rendered. For common expressions, default visual presentations are usually clear. ‘Take care of the sense and the sounds will take care of themselves’ wrote Lewis Carroll [Carroll1871]. Default presentations are included in the detailed description of each element occurring in Section 4.4.

To accomplish these goals, the MathML content encoding is based on the concept of an expression tree. A content expression tree is constructed from a collection of more primitive objects, referred to herein as *containers* and *operators*. MathML possesses a rich set of predefined container and operator objects, as well as constructs for combining containers and operators in mathematically meaningful ways. The syntax and usage of these content elements and constructions is described in the next section.

4.2 Content Element Usage Guide

Since the intent of MathML content markup is to encode mathematical expressions in such a way that the mathematical structure of the expression is clear, the syntax and usage of content markup must be consistent enough to facilitate automated semantic interpretation. There must be no doubt when, for example, an actual sum, product or function application is intended and if specific numbers are present, there must be enough information present to reconstruct the correct number for purposes of computation. Of course, it is still up to a MathML-compliant processor to decide what is to be done with such a content-based expression, and computation is only one of many options. A renderer or a structured editor might simply use the data and its own built-in knowledge of mathematical structure to render the object. Alternatively, it might manipulate the object to build a new mathematical object. A more computationally oriented system might attempt to carry out the indicated operation or function evaluation.

The purpose of this section is to describe the intended, consistent usage. The requirements involve more than just satisfying the syntactic structure specified by an XML DTD. Failure to conform to the usage as described below will result in a MathML error, even though the expression may be syntactically valid according to the DTD.

In addition to the usage information contained in this section, Section 4.4 gives a complete listing of each content element, providing reference information about their attributes, syntax, examples and suggested default semantics and renderings. The rules for using presentation markup within content markup are explained in Section 5.2.3. An informal EBNF grammar describing the syntax for the content markup is given in Appendix B.

4.2.1 Overview of Syntax and Usage

MathML content encoding is based on the concept of an expression tree. As a general rule, the terminal nodes in the tree represent basic mathematical objects, such as numbers, variables, arithmetic operations and so on. The internal nodes in the tree generally represent some kind of function application or other mathematical construction that builds up a compound object. Function application provides the most important example; an internal node might represent the application of a function to several arguments, which are themselves represented by the terminal nodes underneath the internal node.

The MathML content elements can be grouped into the following categories based on their usage:

- containers
- operators and functions
- qualifiers
- relations
- conditions
- semantic mappings
- constants and symbols

These are the building blocks out of which MathML content expressions are constructed. Each category is discussed in a separate section below. In the remainder of this section, we will briefly introduce some of the most common elements of each type, and consider the general constructions for combining them in mathematically meaningful ways.

4.2.1.1 Constructing Mathematical Objects

Content expression trees are built up from basic mathematical objects. At the lowest level, *leaf nodes* are encapsulated in non-empty elements that define their type. Numbers and symbols are marked by the *token* elements `cn` and `ci`. More elaborate constructs such as sets, vectors and matrices are also marked using elements to denote their types, but rather than containing data directly, these *container* elements are constructed out of other elements. Elements are used in order to clearly identify the underlying objects. In this way, standard XML parsing can be used and attributes can be used to specify global properties of the objects.

The containers such as `<cn>12345<cn/>` , `<ci>x</ci>` and `<csymbol definitionURL="mySymbol.htm" encoding="text">S</csymbol>`represent mathematical numbers , identifiers and externally defined symbols. Below, we will look at `operator` elements such as `plus` or `sin`, which provide access to the basic mathematical operations and functions applicable to those objects. Additional containers such as `set` for sets, and `matrix` for matrices are provided for representing a variety of common compound objects.

For example, the number 12345 is encoded as

```
<cn>12345</cn>
```

The attributes and `PCDATA` content together provide the data necessary for an application to parse the number. For example, a default base of 10 is assumed, but to communicate that the underlying data was actually written in base 8, simply set the `base` attribute to 8 as in

```
<cn base="8">12345</cn>
```

while the complex number 3 + 4i can be encoded as

```
<cn type="complex">3<sep/>4</cn>
```

Such information makes it possible for another application to easily parse this into the correct number.

As another example, the scalar symbol v is encoded as

```
<ci>v</ci>
```

By default, ci elements represent elements from a commutative field (see Appendix C). If a vector is intended then this fact can be encoded as

```
<ci type="vector">v</ci>
```

This invokes default semantics associated with the vector element, namely an arbitrary element of a finite-dimensional vector space.

By using the ci and csymbol elements we have made clear that we are referring to a mathematical identifier or symbol but this does not say anything about how it should be rendered. By default a symbol is rendered as if the ci or csymbolelement were actually the presentation element mi (see Section 3.2.2). The actual rendering of a mathematical symbol can be made as elaborate as necessary simply by using the more elaborate presentational constructs (as described in Chapter 3) in the body of the ci or csymbol element.

The default rendering of a simple cn-tagged object is the same as for the presentation element mn with some provision for overriding the presentation of the PCDATA by providing explicit mntags. This is described in detail in Section 4.4.

The issues for compound objects such as sets, vectors and matrices are all similar to those outlined above for numbers and symbols. Each such object has global properties as a mathematical object that impact how they are to be parsed. This may affect everything from the interpretation of operations that are applied to them through to how to render the symbols representing them. These mathematical properties are captured by setting attribute values.

4.2.1.2 Constructing General Expressions

The notion of constructing a general expression tree is essentially that of applying an operator to sub-objects. For example, the sum $a + b$ can be thought of as an application of the addition operator to two arguments a and b. In MathML, elements are used for operators for much the same reason that elements are used to contain objects. They are recognized at the level of XML parsing, and their attributes can be used to record or modify the intended semantics. For example, with the MathML plus element, setting the definitionURL and encodingattributes as in

```
<plus definitionURL="www.vnbooks.com/VectorCalculus.htm"
      encoding="text"/>
```

can communicate that the intended operation is vector-based.

There is also another reason for using elements to denote operators. There is a crucial semantic distinction between the function itself and the expression resulting from applying that function to zero or more arguments which must be captured. This is addressed by making the functions self-contained objects with their own properties and providing an explicit apply construct corresponding to function application. We will consider the apply construct in the next section.

MathML contains many pre-defined operator elements, covering a range of mathematical subjects. However, an important class of expressions involve unknown or user-defined functions and symbols. For these situations, MathML provides a general csymbol element, which is discussed below.

4.2.1.3 The apply construct

The most fundamental way of building up a mathematical expression in MathML content markup is the apply construct. An apply element typically applies an operator to its arguments. It corresponds to a complete mathematical expression. Roughly speaking, this means a piece of mathematics that could be surrounded by parentheses or ‘logical brackets’ without changing its meaning.

For example, $(x + y)$ might be encoded as

```
<apply>
  <plus/>
  <ci> x </ci>
  <ci> y </ci>
</apply>
```

The opening and closing tags of `apply` specify exactly the scope of any operator or function. The most typical way of using `apply` is simple and recursive. Symbolically, the content model can the described as:

```
<apply>
 op
 a
 b  </apply>
```

where the *operands* a and b are containers or other content-based elements themselves, and *op* is an operator or function. Note that since `apply` is a container, this allows `apply` constructs to be nested to arbitrary depth.

An `apply` may in principle have any number of operands:

```
<apply> op a b [c...] <apply>
```

For example, $(x + y + z)$ can be encoded as

```
<apply>
  <plus/>
  <ci> a </ci>
  <ci> b </ci>
  <ci> c </ci>
</apply>
```

Mathematical expressions involving a mixture of operations result in nested occurrences of `apply`. For example, $a\,x + b$ would be encoded as

```
<apply>
  <plus/>
  <apply>
    <times/>
    <ci> a </ci>
    <ci> x </ci>
  </apply>
  <ci> b </ci>
</apply>
```

There is no need to introduce parentheses or to resort to operator precedence in order to parse the expression correctly. The `apply` tags provide the proper grouping for the re-use of the expressions within other constructs. Any expression enclosed by an `apply` element is viewed as a single coherent object.

An expression such as $(F + G)(x)$ might be a product, as in

```
<apply>
  <times/>
  <apply>
    <plus/>
    <ci> F </ci>
    <ci> G </ci>
  </apply>
  <ci> x </ci>
</apply>
```

or it might indicate the application of the function $F + G$ to the argument x. This is indicated by constructing the sum

```
<apply>
  <plus/>
  <ci> F </ci>
  <ci> G </ci>
</apply>
```

and applying it to the argument x as in

```
<apply>
  <apply>
    <plus/>
    <ci> F </ci>
    <ci> G </ci>
    </apply>
  <ci> x </ci>
</apply>
```

Both the function and the arguments may be simple identifiers or more complicated expressions.

In MathML 1.0 , another construction closely related to the use of the `apply` element with operators and arguments was the `reln` element. The `reln`element was used to denote that a mathematical relation holds between its arguments, as opposed to applying an operator. Thus, the MathML markup for the expression $x < y$ was given in MathML 1.0 by:

```
<reln>
  <lt/>
  <ci> x </ci>
  <ci> y </ci>
</reln>
```

In MathML 2.0, the `apply` construct is used with all operators, including logical operators. The expression above becomes

```
<apply>
  <lt/>
  <ci> x </ci>
  <ci> y </ci>
</apply>
```

in MathML 2.0. The use of `reln` with relational operators is supported for reasons of backwards compatibility, but deprecated. Authors creating new content are encouraged to use `apply` in all cases.

4.2.1.4 Explicitly defined functions and operators

The most common operations and functions such as `plus` and `sin` have been predefined explicitly as empty elements (see Section 4.4). They have `type` and `definitionURL`attributes, and by changing these attributes, the author can record that a different sort of algebraic operation is intended. This allows essentially the same notation to be re-used for a discussion taking place in a different algebraic domain.

Due to the nature of mathematics the notation must be extensible. The key to extensibility is the ability of the user to define new functions and other symbols to expand the terrain of mathematical discourse.

It is always possible to create arbitrary expressions, and then to use them as symbols in the language. Their properties can then be inferred directly from that usage as was done in the previous section. However, such an approach would preclude being able to encode the fact that the construct was a known symbol, or to record its mathematical properties except by actually using it. The csymbol element is used as a container to construct a new symbol in much the same way that ci is used to construct an identifier. (Note that 'symbol' is used here in the abstract sense and has no connection with any presentation of the construct on screen or paper). The difference in usage is that csymbol should refer to some mathematically defined concept with an external definition referenced via the definitionURL attribute, whereas ci is used for identifiers that are essentially 'local' to the MathML expression and do not use any external definition mechanism. The target of the definitionURLattribute on the csymbol element may encode the definition in any format: the particular encoding in use is given by the encoding attribute

To use csymbol to describe a completely new function, we write for example

```
<csymbol definitionURL="www.vnbooks.com/VectorCalculus.htm"
         encoding="text">
  Christoffel
</csymbol>
```

The definitionURL attribute specifies a URI that provides a written definition for the Christoffel symbol. Suggested default definitions for the content elements of MathML appear in Appendix C in a format based on OpenMath, although there is no requirement that a particular format be used. The role of the definitionURL attribute is very similar to the role of definitions included at the beginning of many mathematical papers, and which often just refer to a definition used by a particular book.

MathML 1.0 supported the use of the fn to encode the fact that a construct is explicitly being used as a function or operator. To record the fact that $F+G$ is being used semantically as if it were a function, it was encoded as:

```
<fn>
  <apply>
    <plus/>
    <ci>F</ci>
    <ci>G</ci>
  </apply>
</fn>
```

This usage, although allowed in MathML 2.0 for reasons of backwards compatibility, is now deprecated. The fact that a construct is being used as an operator is clear from the position of the construct as the first child of the apply. If it is required to add additional information to the construct, it should be wrapped in a semanticselement, for example:

```
<semantics definitionURL="www.mathslib.com/vectorfuncs/plus.htm"
           encoding="Mathematica">
  <apply>
    <plus/>
    <ci>F</ci>
    <ci>G</ci>
  </apply>
</semantics>
```

MathML 1.0 supported the use of definitionURL with fn to refer to external definitions for user-defined functions. This usage, although allowed for reasons of backwards compatibility, is deprecated in MathML 2.0 in favour of using csymbol to define the function, and then apply to link the function to its arguments. For example:

```
<apply>
  <csymbol definitionURL="http://www.defs.org/function_spaces.html#my_def"
           encoding="text">
    BigK
  </csymbol>
  <ci>x</ci>
  <ci>y</ci>
</apply>
```

4.2.1.5 The inverse construct

Given functions, it is natural to have functional inverses. This is handled by the `inverse` element.

Functional inverses can be problematic from a mathematical point of view in that it implicitly involves the definition of an inverse for an arbitrary function F. Even at the K-through-12 level the concept of an inverse F^{-1} of many common functions $F$is not used in a uniform way. For example, the definitions used for the inverse trigonometric functions may differ slightly depending on the choice of domain and/or branch cuts.

MathML adopts the view: if F is a function from a domain D to D', then the inverse G of F is a function over D' such that $G(F(x)) = x$ for x in D. This definition does not assert that such an inverse exists for all or indeed any x in D, or that it is single-valued anywhere. Also, depending on the functions involved, additional properties such as $F(G(y)) = y$ for y in D' may hold.

The `inverse` element is applied to a function whenever an inverse is required. For example, application of the inverse sine function to x, i.e. $\sin^{-1}(x)$, is encoded as:

```
<apply>
  <apply> <inverse/> <sin/> </apply>
  <ci> x </ci>
</apply>
```

While `arcsin` is one of the predefined MathML functions, an explicit reference to $\sin^{-1}(x)$ might occur in a document discussing possible definitions of `arcsin`.

4.2.1.6 The declare construct

Consider a document discussing the vectors $A = (a, b, c)$ and $B = (d, e, f)$, and later including the expression $V = A + B$. It is important to be able to communicate the fact that wherever A and $B$are used they represent a particular vector. The properties of that vector may determine aspects of operators such as `plus`.

The simple fact that A is a vector can be communicated by using the markup

```
<ci type="vector">A</ci>
```

but this still does not communicate, for example, which vector is involved or its dimensions.

The `declare` construct is used to associate specific properties or meanings with an object. The actual declaration itself is not rendered visually (or in any other form). However, it indirectly impacts the semantics of all affected uses of the declared object.

The scope of a declaration is, by default, local to the MathML element in which the declaration is made. If the `scope`attribute of the `declare` element is set to `global`, the declaration applies to the entire MathML expression in which it appears.

The uses of the `declare` element range from resetting default attribute values to associating an expression with a particular instance of a more elaborate structure. Subsequent uses of the original expression (within the scope of the `declare`) play the same semantic role as would the paired object.

For example, the declaration

```
<declare>
  <ci> A </ci>
  <vector>
    <ci> a </ci>
    <ci> b </ci>
    <ci> c </ci>
  </vector>
</declare>
```

specifies that A stands for the particular vector (a, b, c) so that subsequent uses of A as in $V = A + B$ can take this into account. When `declare` is used in this way, the actual encoding

```
<apply>
  <eq/>
  <ci> V </ci>
  <apply>
    <plus/>
    <ci> A </ci>
    <ci> B </ci>
  </apply>
</apply>
```

remains unchanged but the expression can be interpreted properly as vector addition.

There is no requirement to declare an expression to stand for a specific object. For example, the declaration

```
<declare type="vector">
  <ci> A </ci>
</declare>
```

specifies that A is a vector without indicating the number of components or the values of specific components. The possible values for the `type` attribute include all the predefined container element names such as `vector`, `matrix` or `set` (see Section 4.3.2.9).

4.2.1.7 The lambda construct

The lambda calculus allows a user to construct a function from a variable and an expression. For example, the lambda construct underlies the common mathematical idiom illustrated here:

Let f be the function taking x to $x^2 + 2$

There are various notations for this concept in mathematical literature, such as $\lambda(x, F(x)) = F$ or $\lambda(x, [F]) = F$, where x is a free variable in F.

This concept is implemented in MathML with the `lambda` element. A lambda construct with n internal variables is encoded by a `lambda` element with n+1 children. All but the last child must be `bvar` elements containing the identifiers of the internal variables. The last child is an expression defining the function. This is typically an `apply`, but can also be any container element.

The following constructs λ $(x, \sin(x+1))$:

```
<lambda>
  <bvar><ci> x </ci></bvar>
  <apply>
    <sin/>
    <apply>
      <plus/>
      <ci> x </ci>
      <cn> 1 </cn>
    </apply>
  </apply>
</lambda>
```

To use `declare` and `lambda` to construct the function f for which $f(x) = x^2 + x + 3$ use:

```
<declare type="fn">
  <ci> f </ci>
  <lambda>
    <bvar><ci> x </ci></bvar>
    <apply>
      <plus/>
      <apply>
        <power/>
        <ci> x </ci>
        <cn> 2 </cn>
      </apply>
      <ci> x </ci>
      <cn> 3 </cn>
    </apply>
  </lambda>
</declare>
```

The following markup declares and constructs the function J such that $J(x, y)$ is the integral from x to y of t^4 with respect to t.

```
<declare type="fn">
  <ci> J </ci>
  <lambda>
    <bvar><ci> x </ci></bvar>
    <bvar><ci> y </ci></bvar>
    <apply> <int/>
      <bvar>
        <ci> t </ci>
      </bvar>
      <lowlimit>
        <ci> x </ci>
      </lowlimit>
      <uplimit>
        <ci> y </ci>
      </uplimit>
      <apply>   <power/>
        <ci>t</ci>
        <cn>4</cn>
      </apply>
    </apply>
  </lambda>
</declare>
```

The function J can then in turn be applied to an argument pair.

4.2.1.8 The use of qualifier elements and the condition construct

The last example of the preceding section illustrates the use of *qualifier* elements `lowlimit`, `uplimit`, and `bvar` used in conjunction with the `int` element. A number of common mathematical constructions involve additional data that is either implicit in conventional notation, such as a bound variable, or thought of as part of the operator rather than an argument, as is the case with the limits of a definite integral.

Content markup uses qualifier elements in conjunction with a number of operators, including integrals, sums, series, and certain differential operators. Qualifier elements appear in the same `apply` element with one of these operators. In general, they must appear in a certain order, and their precise meaning depends on the operators being used. For details, see Section 4.2.3.2.

The qualifier element `bvar` is also used in another important MathML construction. The `condition` element is used to place conditions on bound variables in other expressions. This allows MathML to define sets by rule, rather than enumeration, for example. The following markup, for instance, encodes the set $x \mid x < 1$:

```
<set>
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply>
      <lt/>
      <ci> x </ci>
      <cn> 1 </cn>
    </apply>
  </condition>
</set>
```

4.2.1.9 Rendering of Content elements

While the primary role of the MathML content element set is to directly encode the mathematical structure of expressions independent of the notation used to present the objects, rendering issues cannot be ignored. Each content element has a default rendering, given in Section 4.4, and several mechanisms (including Section 4.3.3.2) are provided for associating a particular rendering with an object.

4.2.2 Containers

Containers provide a means for the construction of mathematical objects of a given type.

Tokens	ci, cn, csymbol
Constructors	interval, list, matrix, matrixrow, set, vector, apply, reln, fn, lambda
Specials	declare

4.2.2.1 Tokens

Token elements are typically the leaves of the MathML expression tree. Token elements are used to indicate mathematical identifiers, numbers and symbols.

It is also possible for the canonically empty operator elements such as exp, sin and cos to be leaves in an expression tree. The usage of operator elements is described in Section 4.2.3.

cn The cn element is the MathML token element used to represent numbers. The supported types of numbers include: real, integer, rational, complex-cartesian, and complex-polar, with real being the default type. An attribute base (with default value 10) is used to help specify how the content is to be parsed. The content itself is essentially PCDATA, separated by <sep/> when two parts are needed in order to fully describe a number. For example, the real number 3 is constructed by <cn type="real"> 3 </cn>, while the rational number 3/4 is constructed as <cn type="rational"> 3<sep/>4 </cn>. The detailed structure and specifications are provided in Section 4.4.1.1.

ci The ci element, or 'content identifier' is used to construct a variable, or an identifier. A type attribute indicates the type of object the symbol represents. Typically, ci represents a real scalar, but no default is specified. The content is either PCDATA or a general presentation construct (see Section 3.1.5). For example,

```
<ci>
<msub>
  <mi>c</mi>
  <mn>1</mn>
</msub>
</ci>
```

encodes an atomic symbol that displays visually as $c_1$which, for purposes of content, is treated as a single symbol representing a real number. The detailed structure and specifications is provided in Section 4.4.1.2.

csymbol The csymbol element, or 'content symbol' is used to construct a symbol whose semantics are not part of the core content elements provided by MathML, but defined externally. csymbol does not make any attempt to describe how to map the arguments occurring in any application of the function into a new MathML expression. Instead, it depends on its definitionURL attribute to point to a particular meaning, and the encoding attribute to give the syntax of this definition. The content of a csymbol is either PCDATAor a general presentation construct (see Section 3.1.5). For example,

```
<csymbol definitionURL="www.vnbooks.com/ContDiffFuncs.htm"
         encoding="text">
<msup>
  <mi>C</mi>
  <mn>2</mn>
</msup>
</csymbol>
```

encodes an atomic symbol that displays visually as C^2 and that, for purposes of content, is treated as a single symbol representing the space of twice-differentiable continuous functions. The detailed structure and specifications is provided in Section 4.4.1.3.

4.2.2.2 Constructors

MathML provides a number of elements for combining elements into familiar compound objects. The compound objects include things like lists, sets. Each constructor produces a new type of object.

interval The interval element is described in detail in Section 4.4.2.4. It denotes an interval on the real line with the values represented by its children as end points. The closure attribute is used to qualify the type of interval being represented. For example,

```
<interval closure="open-closed">
  <ci> a </ci>
  <ci> b </ci>
</interval>
```

represents the open-closed interval often written $(a, b]$.

set and list The set and list elements are described in detail in Section 4.4.6.1 and Section 4.4.6.2. Typically, the child elements of a possibly empty list element are the actual components of an ordered *list*. For example, an ordered list of the three symbols a, b, and c is encoded as

```
<list> <ci> a </ci> <ci> b </ci> <ci> c </ci> </list>
```

Alternatively, bvar and condition elements can be used to define lists where membership depends on satisfying certain conditions. An order attribute, which is used to specify what ordering is to be used. When the nature of the child elements permits, the ordering defaults to a numeric or lexicographic ordering. Sets are structured much the same as lists except that there is no implied ordering and the type of set may be normal or multiset with multiset indicating that repetitions are allowed. For both sets and lists, the child elements must be valid MathML content elements. The type of the child elements is not restricted. For example, one might construct a list of equations, or inequalities.

matrix and matrixrow The matrix element is used to represent mathematical matrices. It is described in detail in Section 4.4.10.2. It has zero or more child elements, all of which are matrixrow elements. These in turn expect zero or more child elements that evaluate to algebraic expressions or numbers. These sub-elements are often real numbers, or symbols as in

```
<matrix>
  <matrixrow> <cn> 1 </cn> <cn> 2 </cn> </matrixrow>
  <matrixrow> <cn> 3 </cn> <cn> 4 </cn> </matrixrow>
</matrix>
```

The matrixrow elements must always be contained inside of a matrix, and all rows in a given matrix must have the same number of elements. Note that the behavior of the matrix and matrixrow elements is substantially different from the mtable and mtr presentation elements.

vector The vector element is described in detail in Section 4.4.10.1. It constructs vectors from an n-dimensional vector space so that its n child elements typically represent real or complex valued scalars as in the three-element vector

```
<vector>
  <apply>
    <plus/>
    <ci> x </ci>
    <ci> y </ci>
  </apply>
  <cn> 3 </cn>
  <cn> 7 </cn>
</vector>
```

apply The apply element is described in detail in Section 4.4.2.1. Its purpose is apply a function or operator to its arguments to produce an an expression representing an element of the range of the function. It is involved in everything from forming sums such as $a + b$ as in

```
<apply>
  <plus/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

through to using the sine function to construct $\sin(a)$ as in

```
<apply>
  <sin/>
  <ci> a </ci>
</apply>
```

or constructing integrals. Its usage in any particular setting is determined largely by the properties of the function (the first child element) and as such its detailed usage is covered together with the functions and operators in Section 4.2.3.

reln The `reln` element is described in detail in Section 4.4.2.2. It was used in MathML 1.0 to construct an expression such as $a = b$, as in

```
<reln><eq/>
  <ci> a </ci>
  <ci> b </ci>
</reln>
```

indicating an intended comparison between two mathematical values. MathML 2.0 takes the view that this should be regarded as the application of a boolean function, and as such could be constructed using `apply`. The use of `reln` with logical operators is supported for reasons of backwards compatibility, but deprecated in favour of `apply`.

fn The `fn` element was used in MathML 1.0 to make explicit the fact that an expression is being used as a function or operator. This is allowed in MathML 2.0 for backwards compatibility, but is deprecated, as the use of an expression as a function or operator is clear from its position as the first child of an `apply`. `fn` is discussed in detail in Section 4.4.2.3.

lambda The `lambda` element is used to construct a user-defined function from an expression and one or more free variables. The lambda construct with n internal variables takes n+1 children. The first (second, up to n) is a `bvar` containing the identifiers of the internal variables. The last is an expression defining the function. This is typically an `apply`, but can also be any container element. The following constructs $\lambda(x, \sin x)$

```
<lambda>
  <bvar><ci> x </ci></bvar>
  <apply>
    <sin/>
    <ci> x </ci>
  </apply>
</lambda>
```

The following constructs the constant function $\lambda(x, 3)$

```
<lambda>
  <bvar><ci> x </ci></bvar>
  <cn> 3 </cn>
</lambda>
```

4.2.2.3 Special Constructs

The `declare` construct is described in detail in Section 4.4.2.8. It is special in that its entire purpose is to modify the semantics of other objects. It is not rendered visually or aurally.

The need for declarations arises any time a symbol (including more general presentations) is being used to represent an instance of an object of a particular type. For example, you may wish to declare that the symbolic identifier V represents a vector.

The declaration

```
<declare type="vector"><ci>V</ci></declare>
```

resets the default type attribute of `<ci>V</ci>` to `vector` for all affected occurrences of `<ci>V</ci>`. This avoids having to write `<ci type="vector">V</ci>` every time you use the symbol.

More generally, `declare` can be used to associate expressions with specific content. For example, the declaration

```
<declare>
  <ci>F</ci>
  <lambda>
    <bvar><ci> U </ci></bvar>
    <apply>
      <int/>
      <bvar><ci> x </ci></bvar>
      <lowlimit><cn> 0 </cn></lowlimit>
      <uplimit><ci> a </ci></uplimit>
      <ci> U </ci>
    </apply>
  </lambda>
</declare>
```

associates the symbol F with a new function defined by the `lambda` construct. Within the scope where the declaration is in effect, the expression

```
<apply>
  <ci>F</ci>
  <ci> U </ci>
</apply>
```

stands for the integral of U from 0 to a.

The `declare` element can also be used to change the definition of a function or operator. For example, if the URL `http://.../MathML:noncommutplus` described a non-commutative plus operation encoded in Maple syntax, then the declaration

```
<declare definitionURL="http://.../MathML:noncommutplus"
         encoding="Maple">
  <plus/>
</declare>
```

would indicate that all affected uses of `plus` are to be interpreted as having that definition of `plus`.

4.2.3 Functions, Operators and Qualifiers

The operators and functions defined by MathML can be divided into categories as shown in the table below.

unary arithmetic	exp, factorial, minus, abs, conjugate, arg, real, imaginary
unary logical	not
unary functional	inverse, ident
unary elementary classical functions	sin, cos, tan, sec, csc, cot, sinh, cosh, tanh, sech, csch, coth, arcsin, arccos, arctan, arccosh, arccot, arccoth, arccs arcsinh, arctanh, exp, ln, log
unary linear algebra	determinant, transpose
unary calculus and vector calculus	divergence, grad, curl, laplacian
unary set-theoretic	card
binary arithmetic	quotient, divide, minus, power, rem
binary logical	implies, equivalent, approx
binary set operators	setdiff
binary linear algebra	vectorproduct, scalarproduct, outerproduct
n-ary arithmetic	plus, times, max, min, gcd, lcm
n-ary statistical	mean, sdev, variance, median, mode
n-ary logical	and, or, xor
n-ary linear algebra	selector
n-ary set operator	union, intersect
n-ary functional	fn, compose
integral, sum, product operators	int, sum, product
differential operator	diff, partialdiff
quantifier	forall, exists

From the point of view of usage, MathML regards functions (for example `sin` and `cos`) and operators (for example `plus` and `times`) in the same way. MathML predefined functions and operators are all canonically empty elements.

Note that the `csymbol` element can be used to construct a user-defined symbol that can be used as a function or operator.

4.2.3.1 Predefined functions and operators

MathML functions can be used in two ways. They can be used as the operator within an `apply` element, in which case they refer to a function evaluated at a specific value. For example,

```
<apply>
  <sin/>
  <cn>5</cn>
</apply>
```

denotes a real number, namely sin(5).

MathML functions can also be used as arguments to other operators, for example

```
<apply>
  <plus/><sin/><cos/>
</apply>
```

denotes a function, namely the result of adding the sine and cosine functions in some function space. (The default semantic definition of `plus` is such that it infers what kind of operation is intended from the type of its arguments.)

The number of child elements in the `apply` is defined by the element in the first (i.e. operator) position.

Unary operators are followed by exactly one other child element within the `apply`.

Binary operators are followed by exactly two child elements.

N-ary operators are followed by zero or more child elements.

The one exception to these rules is that `declare` elements may be inserted in any position except the first. `declare` elements are not counted when satisfying the child element count for an `apply`containing a unary or binary operator element.

Integral, sum, product and differential operators are discussed below in Section 4.2.3.2.

4.2.3.2 Operators taking Qualifiers

The table below contains the qualifiers and the operators taking qualifiers in MathML.

qualifiers	`lowlimit`, `uplimit`, `bvar`, `degree`, `logbase`, `interval`, `condition`
operators	`int`, `sum`, `product`, `root`, `diff`, `partialdiff`, `limit`, `log`, `moment`, `min`, `max`, `forall`, `exists`

Operators taking qualifiers are canonically empty functions that differ from ordinary empty functions only in that they support the use of special *qualifier* elements to specify their meaning more fully. They are used in exactly the same way as ordinary operators, except that when they are used as operators, certain qualifier elements are also permitted to be in the enclosing `apply`. They always precede the argument if it is present. If more than one qualifier is present, they appear in the order `bvar`, `lowlimit`, `uplimit`, `interval`, `condition`, `degree`, `logbase`. A typical example is:

```
<apply>
  <int/>
  <bvar><ci>x</ci></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <uplimit><cn>1</cn></uplimit>
  <apply>
      <power/>
    <ci>x</ci>
    <cn>2</cn>
  </apply>
</apply>
```

It is also valid to use qualifier schema with a function not applied to an argument. For example, a function acting on integrable functions on the interval [0,1] might be denoted:

```
<fn>
  <apply>
    <int/>
    <bvar><ci>x</ci></bvar>
    <lowlimit><cn>0</cn></lowlimit>
    <uplimit><cn>1</cn></uplimit>
  </apply>
</fn>
```

The meaning and usage of qualifier schema varies from function to function. The following list summarizes the usage of qualifier schema with the MathML functions taking qualifiers.

int The `int` function accepts the `lowlimit`, `uplimit`, `bvar`, `interval` and `condition` schemata. If both `lowlimit` and `uplimit` schema are present, they denote the limits of a definite integral. The domain of integration may alternatively be specified using interval or condition. The `bvar` schema signifies the variable of integration. When used with `int`, each qualifier schema is expected to contain a single child schema; otherwise an error is generated.

diff The `diff` function accepts the `bvar` schema. The `bvar` schema specifies with respect to which variable the derivative is being taken. The `bvar` may itself contain a `degree` schema that is used to specify the order of the derivative, i.e. a first derivative, a second derivative, etc. For example, the second derivative of f with respect to x is:

```
<apply>
  <diff/>
  <bvar>
    <ci> x </ci>
    <degree>
      <cn> 2 </cn>
    </degree>
  </bvar>
  <apply><fn><ci>f</ci></fn>
    <ci> x </ci>
  </apply>
</apply>
```

partialdiff The partialdiff function accepts zero or more bvar schemata. The bvarschema specify with respect to which variables the derivative is being taken. The bvar elements may themselves contain degree schemata that are used to specify the order of the derivative. Variables specified by multiple bvar elements will be used in order as the variable of differentiation in mixed partials. When used with partialdiff, the degree schema is expected to contain a single child schema. For example,

```
<apply>
  <partialdiff/>
  <bvar><ci>x</ci></bvar>
  <bvar><ci>y</ci></bvar>
  <fn><ci>f</ci></fn>
</apply>
```

denote the mixed partial $(d^2 / d\, x d\, y)\ f$.

sum, product The sum and productfunctions accept the bvar, lowlimit, uplimit, interval and conditionschemata. If both lowlimit and uplimit schemata are present, they denote the limits of the sum or product. The limits may alternatively be specified using the interval or conditionschema. The bvar schema signifies the index variable in the sum or product. A typical example might be:

```
<apply>
  <sum/>
  <bvar><ci>i</ci></bvar>
  <lowlimit><cn>0</cn></lowlimit>
  <uplimit><cn>100</cn></uplimit>
  <apply>
    <power/>
    <ci>x</ci>
    <ci>i</ci>
  </apply>
</apply>
```

When used with sum or product, each qualifier schema is expected to contain a single child schema; otherwise an error is generated.

limit The limit function accepts zero or more bvar schemata, and optional condition and lowlimitschemata. A condition may be used to place constraints on the bvar. The bvar schema denotes the variable with respect to which the limit is being taken. The lowlimit schema denotes the limit point. When used with limit, the bvar and lowlimit schemata are expected to contain a single child schema; otherwise an error is generated.

log The log function accepts only the logbase schema. If present, the logbase schema denotes the base with respect to which the logarithm is being taken. Otherwise, the log is assumed to be base 10. When used with log, the logbase schema is expected to contain a single child schema; otherwise an error is generated.

moment The moment function accepts only the degree schema. If present, the degree schema denotes the order of the moment. Otherwise, the moment is assumed to be the first order moment. When used with moment, the degree schema is expected to contain a single child schema; otherwise an error is generated.

min, max The `min` and `max`functions accept a `bvar` schema in cases where the maximum or minimum is being taken over a set of values specified by a `condition` schema together with an expression to be evaluated on that set. In MathML1.0, the `bvar` element was optional when using a `condition`; if a `condition` element containing a single variable was given by itself following a `min` or `max` operator, the variable was implicitly assumed to be bound, and the expression to be maximized or minimized (if absent) was assumed to be the single bound variable. This usage is deprecated in MathML 2.0 in favour of explicitly stating the bound variable(s) and the expression to be maximised in all cases. The `min` and `max` elements may also be applied to a list of values in which case no qualifier schemata are used. For examples of all three usages, see Section 4.4.3.4.

forall, exists The universal and existential quantifier operators `forall` and `exists` are used in conjuction with one or more `bvar` schemata to represent simple logical assertions. There are two ways of using the logical quantifier operators. The first usage is for representing a simple, quantified assertion. For example, the statement 'there exists $x < 9$' would be represented as:

```
<apply>
  <exists/>
  <bvar><ci> x </ci></bvar>
  <apply><lt/>
    <ci> x </ci><cn> 9 </cn>
  </apply>
</apply>
```

The second usage is for representing implications. Hypotheses are given by a `condition` element following the bound variables. For example the statement 'for all $x < 9$, $x < 10$' would be represented as:

```
<apply>
  <forall/>
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply><lt/>
      <ci> x </ci><cn> 9 </cn>
    </apply>
  </condition>
  <apply><lt/>
    <ci> x </ci><cn> 10 </cn>
  </apply>
</apply>
```

Note that in both usages one or more `bvar` qualifiers are mandatory.

4.2.4 Relations

binary relation	`neq`, `equivalent`, `approx`
binary logical relation	`implies`
binary set relation	`in`, `notin`, `notsubset`, `notprsubset`
binary series relation	`tendsto`
n-ary relation	`eq`, `leq`, `lt`, `geq`, `gt`
n-ary set relation	`subset`, `prsubset`

The MathML content tags include a number of canonically empty elements which denote arithmetic and logical relations. Relations are characterized by the fact that, if an external application were to evaluate them (MathML does not specify how to evaluate expressions), they would typically return a truth value. By contrast, operators generally return a value of the same type as the operands. For example, the result of evaluating $a < b$ is either true or false (by contrast, 1 + 2 is again a number).

Relations are bracketed with their arguments using the `apply` element in the same way as other functions. In MathML 1.0, relational operators were bracketed using `reln`. This usage, although still supported, is now deprecated in favour of `apply`. The element for the relational operator is the first child element of the `apply`. Thus, the example from the preceding paragraph is properly marked up as:

```
<apply>
  <lt/>
  <ci>a</ci>
  <ci>b</ci>
</apply>
```

It is an error to enclose a relation in an element other than `apply` or `reln`.

The number of child elements in the `apply` is defined by the element in the first (i.e. relation) position.

Unary relations are followed by exactly one other child element within the `apply`.

Binary relations are followed by exactly two child elements.

N-ary relations are followed by zero or more child elements.

The one exception to these rules is that `declare` elements may be inserted in any position except the first. `declare` elements are not counted when satisfying the child element count for an `apply`containing a unary or binary relation element.

4.2.5 Conditions

condition `condition`

The `condition` element is used to define the 'such that' construct in mathematical expressions. Condition elements are used in a number of contexts in MathML. They are used to construct objects like sets and lists by rule instead of by enumeration. They can be used with the `forall` and `exists` operators to form logical expressions. And finally, they can be used in various ways in conjunction with certain operators. For example, they can be used with and `int` element to specify domains of integration, or to specify argument lists for operators like `min` and `max`.

The `condition` element is always used together with one or more `bvar` elements.

The exact interpretation depends on the context, but generally speaking, the `condition` element is used to restrict the permissible values of a bound variable appearing in another expression to those that satisfy the relations contained in the `condition`. Similarly, when the `condition` element contains a `set`, the values of the bound variables are restricted to that set.

A condition element contains a single child that is either a `apply`, or a `reln` element (deprecated). Compound conditions are indicated by applying relations such as `and` inside the child of the condition.

4.2.5.1 Examples

The following encodes 'there exists x such that $x^5 < 3$'.

```
<apply>
  <exists/>
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply><lt/>
      <apply>
        <power/>
        <ci>x</ci>
        <cn>5</cn>
      </apply>
      <cn>3</cn>
    </apply>
  </condition>
</apply>
```

The next example encodes ‘for all x in N there exists prime numbers p, q such that $p+q=2x$’.

```
<apply>
  <forall/>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><in/>
          <ci>x</ci>
          <csymbol encoding="text" definitionURL="www.naturalnums.htm">N</csymbol>
        </apply>
  </condition>

  <apply><exists/>
     <bvar><ci>p</ci></bvar>
     <bvar><ci>q</ci></bvar>
     <condition>
       <apply><and/>
          <apply><in/><ci>p</ci>
            <csymbol encoding="text" definitionURL="www.primes.htm">P</csymbol>
          </apply>
          <apply><in/><ci>q</ci>
            <csymbol encoding="text" definitionURL="www.primes.htm">P</csymbol>
          </apply>
          <apply><eq/>
              <apply><plus/><ci>p</ci><ci>q</ci></apply>
              <apply><times/><cn>2</cn><ci>x</ci></apply>
          </apply>
        </apply>
      </condition>
  </apply>
</apply>
```

A third example shows the use of quantifiers with `condition`. The following markup encodes ‘there exists $x < 3$ such that $x^2 = 4$’.

```
<apply>
  <exists/>
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply><lt/><ci>x</ci><cn>3</cn></apply>
  </condition>
  <apply>
    <eq/>
    <apply>
      <power/><ci>x</ci><cn>2</cn>
    </apply>
    <cn>4</cn>
  </apply>
</apply>
```

4.2.6 Syntax and Semantics

mappings `semantics`, `annotation`, `annotation-xml`

The use of content markup rather than presentation markup for mathematics is sometimes referred to as *semantic tagging* [Buswell1996]. The parse-tree of a valid element structure using MathML content elements corresponds directly to the expression tree of the underlying mathematical expression. We therefore regard the content tagging itself as encoding the *syntax* of the mathematical expression. This is, in general, sufficient to obtain some rendering and even some symbolic manipulation (e.g. polynomial factorization).

However, even in such apparently simple expressions as $X + Y$, some additional information may be required for applications such as computer algebra. Are X and Y integers, or functions, etc.? 'Plus' represents addition over which field? This additional information is referred to as *semantic mapping*. In MathML, this mapping is provided by the `semantics`, `annotation` and `annotation-xml` elements.

The `semantics` element is the container element for the MathML expression together with its semantic mappings. `semantics` expects a variable number of child elements. The first is the element (which may itself be a complex element structure) for which this additional semantic information is being defined. The second and subsequent children, if any, are instances of the elements `annotation` and/or `annotation-xml`.

The `semantics` tags also accepts the `definitionURL` and `encoding` attributes for use by external processing applications. One use might be a URI for a semantic content dictionary, for example. Since the semantic mapping information might in some cases be provided entirely by the `definitionURL`attribute, the `annotation` or `annotation-xml` elements are optional.

The `annotation` element is a container for arbitrary data. This data may be in the form of text, computer algebra encodings, C programs, or whatever a processing application expects. `annotation` has an attribute `encoding` defining the form in use. Note that the content model of `annotation` is `PCDATA`, so care must be taken that the particular encoding does not conflict with XML parsing rules.

The `annotation-xml` element is a container for semantic information in well-formed XML. For example, an XML form of the OpenMath semantics could be given. Another possible use here is to embed, for example, the presentation tag form of a construct given in content tag form in the first child element of `semantics` (or vice versa). `annotation-xml` has an attribute `encoding` defining the form in use.

For example:

```
<semantics>
  <apply>
  <divide/>
    <cn>123</cn>
    <cn>456</cn>
  </apply>
  <annotation encoding="Mathematica">
    N[123/456, 39]
  </annotation>
  <annotation encoding="TeX">
    $0.269736842105263157894736842105263157894\ldots$
  </annotation>
  <annotation encoding="Maple">
    evalf(123/456, 39);
  </annotation>
  <annotation-xml encoding="MathML-Presentation">
    <mrow>
      <mn> 0.269736842105263157894 </mn>
      <mover accent='true'>
        <mn> 736842105263157894 </mn>
        <mo> &OverBar; </mo>
      </mover>
    </mrow>
  </annotation-xml>
  <annotation-xml encoding="OpenMath">
    <OMA>...</OMA>
  </annotation-xml>
</semantics>
```

where `OMA` is the element defining the additional semantic information.

Of course, providing an explicit semantic mapping at all is optional, and in general would only be provided where there is some requirement to process or manipulate the underlying mathematics.

4.2.7 Semantic Mappings

Although semantic mappings can easily be provided by various proprietary, or highly specialized encodings, there are no widely available, non-proprietary standard schemes for semantic mapping. In part to address this need, the goal of the OpenMath effort is to provide a platform-independent, vendor-neutral standard for the exchange of mathematical objects between applications. Such mathematical objects include semantic mapping information. The OpenMath group has defined an SGML syntax for the encoding of this information [OpenMath1996]. This element set could provide the basis of one `annotation-xml` element set.

An attractive side of this mechanism is that the OpenMath syntax is specified in XML, so that a MathML expression together with its semantic annotations can be validated using XML parsers.

4.2.8 Constants and Symbols

MathML provdies a collection of predefined constants and symbols which represent frequently-encountered concepts in K-12 mathematics. These include symbols for well-known sets, such as integers `integers` and `rationals`, and also some widely known constant symbols such as `false`, `true`, `exponentiale`.

4.2.9 MathML element types

MathML functions, operators and relations can all be thought of as mathematical functions if viewed in a sufficiently abstract way. For example, the standard addition operator can be regarded as a function mapping pairs of real numbers to real numbers. Similarly, a relation can be thought of as a function from some space of ordered pairs into the set of values true, false. To be mathematically meaningful, the domain and range of a function must be precisely specified. In practical terms, this means that functions only make sense when applied to certain kinds of operands. For example, thinking of the standard addition operator, it makes no sense to speak of 'adding' a set to a function. Since MathML content markup seeks to encode mathematical expressions in a way that can be unambiguously evaluated, it is no surprise that the types of operands is an issue.

MathML specifies the types of arguments in two ways. The first way is by providing precise instructions for processing applications about the kinds of arguments expected by the MathML content elements denoting functions, operators and relations. These operand types are defined in a dictionary of default semantic bindings for content elements, which is given in Appendix C. For example, the MathML content dictionary specifies that for real scalar arguments the plus operator is the standard commutative addition operator over a field. The elements `cn` has a `type` attribute with a default value of `real`. Thus some processors will be able to use this information to verify the validity of the indicated operations.

Although MathML specifies the types of arguments for functions, operators and relations, and provides a mechanism for typing arguments, a MathML-compliant processor is not required to do any type checking. In other words, a MathML processor will not generate errors if argument types are incorrect. If the processor is a computer algebra system, it may be unable to evaluate an expression, but no MathML error is generated.

4.3 Content Element Attributes

4.3.1 Content Element Attribute Values

Content element attributes are all of the type `CDATA`, that is, any character string will be accepted as valid. In addition, each attribute has a list of predefined values, which a content processor is expected to recognize and process. The reason that the attribute values are not formally restricted to the list of predefined values is to allow for extension. A processor encountering a value (not in the predefined list) which it does not recognize may validly process it as the default value for that attribute.

4.3.2 Attributes Modifying Content Markup Semantics

Each attribute is followed by the elements to which it can be applied.

4.3.2.1 `base`

cn indicates numerical base of the number. Predefined values: any numeric string. The default value is `10`

4.3.2.2 `closure`

interval indicates closure of the interval. Predefined values: `open`, `closed`, `open-closed`, `closed-open`. The default value is `closed`

4.3.2.3 `definitionURL`

csymbol, declare, semantics, any operator element points to an external definition of the semantics of the symbol or construct being declared. The value is a URL or URI that should point to some kind of definition. This definition overrides the MathML default semantics. At present, MathML does not specify the format in which external semantic definitions should be given. In particular, *there is no requirement that the target of the URI be loadable and parsable.*An external definition could, for example, define the semantics in human-readable form. Ideally, in most situations the definition pointed to by the `definitionURL` attribute would be some standard, machine-readable format. However, there are several reasons why MathML does not require such a format. First, no such format currently exists. There are several projects underway to develop and implement standard semantic encoding formats, most notably the OpenMath effort. But by nature, the development of a comprehensive system of semantic encoding is a very large enterprise, and while much work has been done, much additional work remains. Therefore, even though the `definitionURL` is designed and intended for use with a formal semantic encoding language such as OpenMath, it is premature to require any one particular format. Another reason for leaving the format of the `definitionURL` attribute unspecified is that there will always be situations where some non-standard format is preferable. This is particularly true in situations where authors are describing new ideas. It is anticipated that in the near term, there will be a variety of renderer-dependent implementations of the `definitionURL` attribute. For example, a translation tool might simply prompt the user with the specified definition in situations where the proper semantics have been overridden, and in this case, human-readable definitions will be most useful. Other software may utilize OpenMath encodings. Still other software may use proprietary encodings, or look for definitions in any of several formats. As a consequence, authors need to be aware that there is no guarantee a generic renderer will be able to take advantage of information pointed to by the `definitionURL` attribute. Of course, when widely-accepted standardized semantic encodings are available, the definitions pointed to can be replaced without modifying the original document. However, this is likely to be labor intensive. There is no default value for the `definitionURL`attribute, i.e. the semantics are defined within the MathML fragment, and/or by the MathML default semantics.

4.3.2.4 `encoding`

annotation, annotation-xml, csymbol, semantics, all operator elements indicates the encoding of the annotation, or in the case of `csymbol` , `semantics` and operator elements, the syntax of the target referred to by `definitionURL`. Predefined values are `MathML-Presentation`, `MathML-Content`. Other typical values: `TeX`, `OpenMath`. The default value is "", i.e. unspecified.

4.3.2.5 `nargs`

declare indicates number of arguments for function declarations. Pre-defined values: `nary`, or any numeric string. The default value is `1`

4.3.2.6 `occurrence`

declare indicates occurrence for operator declarations. Pre-defined values: `prefix`, `infix`, `function-model`. The default value is `function-model`

4.3.2.7 `order`

list indicates ordering on the list. Predefined values: `lexicographic`, `numeric`. The default value is `numeric`

4.3.2.8 `scope`

declare indicates scope of applicability of the declaration. Pre-defined values: `local`, `global`.

- `local` means the containing MathML element.
- `global` means the containing `math` element.

The default value is `local`. At present, declarations cannot affect anything outside of the containing `math` element. Ideally, one would like to make document-wide declarations by setting the value of the `scope` attribute to be `global-document`. However, the proper mechanism for document-wide declarations very much depends on details of the way in which XML will be embedded in HTML, future XML style sheet mechanisms, and the underlying Document Object Model. Since these supporting technologies are still in flux at present, the MathML specification does not include `global-document` as a pre-defined value of the `scope` attribute. It is anticipated, however, that this issue will be revisited in future revisions of MathML as supporting technologies stabilize. In the near term, MathML implementors that wish to simulate the effect of a document-wide declaration are encouraged to pre-process documents in order to distribute document-wide declarations to each individual `math` element in the document.

4.3.2.9 type

cn indicates type of the number. Predefined values: integer, rational, real, float, complex, complex-polar, complex-cartesian, constant. The default value is real. Notes. Each data type implies that the data adheres to certain formating conventions, detailed below. If the data fails to conform to the expected format, an error is generated. Details of the individual formats are:

real A real number is presented in decimal notation. Decimal notation consists of an optional sign ('+'or '-') followed by a string of digits possibly separated into an integer and a fractional part by a 'decimal point'. Some examples are 0.3, 1, and -31.56. If a different base is specified, then the digits are interpreted as being digits computed to that base. A real number may also be presented in scientific notation. Such numbers have two parts (a mantissa and an exponent) separated by 'e'. The first part is a real number, while the second part is an integer exponent indicating a power of the base. For example, 12.3e5 represents 12.3 times 10^5.

integer An integer is represented by an optional sign followed by a string of 1 or more 'digits'. What a 'digit' is depends on the baseattribute. If base is present, it specifies the base for the digit encoding, and it specifies it base ten. Thus base='16' specifies a hex encoding. When base > 10, letters are added in alphabetical order as digits. The legitimate values for base are therefore between 2 and 36.

rational A rational number is two integers separated by <sep/>. If base is present, it specifies the base used for the digit encoding of both integers.

complex-cartesian A complex number is of the form two real point numbers separated by <sep/>.

complex-polar A complex number is specified in the form of a magnitude and an angle (in radians). The raw data is in the form of two real numbers separated by <sep/>.

constant The constant type is used to denote named constants. For example, an instance of <cn type="constant">π</cn>should be interpreted as having the semantics of the mathematical constant Pi. The data for a constant cn tag may be one of the following common constants:

Symbol	Value
π	The usual π of trigonometry: approximately 3.141592653...
ⅇ (or ⅇ)	The base for natural logarithms: approximately 2.718281828 ...
ⅈ (or ⅈ)	Square root of -1
γ	Euler's constant: approximately 0.5772156649...
∞ (or &infty;)	Infinity. Proper interpretation varies with context
&true;	the logical constant true
&false;	the logical constant false
&NotANumber; (or &NaN;)	represents the result of an ill-defined floating point division

ci indicates type of the identifier. Predefined values: integer, rational, real, float, complex, complex-polar, complex-cartesian, constant, or the name of any content element. The meaning of the various attribute values is the same as that listed above for the cn element. The default value is "", i.e. unspecified.

declare indicates type of the identifier being declared. Predefined values: any content element name. The default value is ci , i.e. a generic identifier

set indicates type of the set. Predefined values: normal, multiset. multiset indicates that repetitions are allowed. The default value is normal.

tendsto indicates the direction from which the limiting value is approached. Predefined values: above, below, two-sided. The default value is above.

4.3.3 Attributes Modifying Content Markup Rendering

4.3.3.1 type

The type attribute, in addition to conveying semantic information, can be interpreted to provide rendering information. For example in

```
<ci type="vector">V</ci>
```

a renderer could display a bold V for the vector.

4.3.3.2 General Attributes

All content elements support the following general attributes that can be used to modify the rendering of the markup.

- class
- style

- `id`
- `other`

The `class`, `style` and `id` attributes are intended for compatibility with Cascading Style Sheets (CSS), as described in Section 2.3.5.

Content or semantic tagging goes along with the (frequently implicit) premise that, if you know the semantics, you can always work out a presentation form. When an author's main goal is to mark up re-usable, evaluatable mathematical expressions, the exact rendering of the expression is probably not critical, provided that it is easily understandable. However, when an author's goal is more along the lines of providing enough additional semantic information to make a document more accessible by facilitating better visual rendering, voice rendering, or specialized processing, controlling the exact notation used becomes more of an issue.

MathML elements accept an attribute `other` (see Section 7.2.3), which can be used to specify things not specifically documented in MathML. On content tags, this attribute can be used by an author to express a *preference* between equivalent forms for a particular content element construct, where the selection of the presentation has nothing to do with the semantics. Examples might be

- inline or displayed equations
- script-style fractions
- use of x with a dot for a derivative over d x/d t

Thus, if a particular renderer recognized a display attribute to select between script-style and display-style fractions, an author might write

```
<apply other='display="scriptstyle"'>
  <divide/>
  <mn> 1 </mn>
  <mi> x </mi>
</apply>
```

to indicate that the rendering 1/ x is preferred.

The information provided in the `other` attribute is intended for use by specific renderers or processors, and therefore, the permitted values are determined by the renderer being used. It is legal for a renderer to ignore this information. This might be intentional, in the case of a publisher imposing a house style, or simply because the renderer does not understand them, or is unable to carry them out.

4.4 The Content Markup Elements

This section provides detailed descriptions of the MathML content tags. They are grouped in categories that broadly reflect the area of mathematics from which they come, and also the grouping in the MathML DTD. There is no linguistic difference in MathML between operators and functions. Their separation here and in the DTD is for reasons of historical usage.

When working with the content elements, it can be useful to keep in mind the following.

- The role of the content elements is analogous to data entry in a mathematical system. The information that is provided is there to facilitate the successful parsing of an expression as the intended mathematical object by a receiving application.
- MathML content elements do not by themselves 'perform' any mathematical evaluations or operations. They do not 'evaluate' in a browser and any 'action' that is ultimately taken on those objects is determined entirely by the receiving mathematical application. For example, editing programs and applications geared to computation for the lower grades would typically leave 3 + 4 as is, whereas computational systems targeting a more advanced audience might evaluate this as 7. Similarly, some computational systems might evaluate sin(0) to 0, whereas others would leave it unevaluated. Yet other computational systems might be unable to deal with pure symbolic expressions sin(x) and may even regard it as a data entry error. None of this has any bearing on the correctness of the original MathML representation. Where evaluation is mentioned at all in the descriptions below, it is merely to help clarify the meaning of the underlying operation.
- Apart from the instances where there is an explicit interaction with presentation tagging, there is no required rendering (visual or aural) - only a suggested default. As such, the presentations that are included in this section are merely to help communicate to the reader the intended mathematical meaning by association with the same expression written in a more traditional notation.

The available content elements are:

- token elements
  - `cn`
  - `ci`
  - `csymbol` (MathML 2.0)
- basic content elements
  - `apply`
  - `reln` (deprecated)
  - `fn` (deprecated for externally defined functions)
  - `interval`
  - `inverse`
  - `sep`
  - `condition`
  - `declare`
  - `lambda`
  - `compose`
  - `ident`
- arithmetic, algebra and logic
  - `quotient`
  - `exp`
  - `factorial`
  - `divide`
  - `max` and `min`
  - `minus`
  - `plus`
  - `power`
  - `rem`
  - `times`
  - `root`
  - `gcd`
  - `and`
  - `or`
  - `xor`
  - `not`
  - `implies`
  - `forall`
  - `exists`
  - `abs`
  - `conjugate`
  - `arg` (MathML 2.0)
  - `real` (MathML 2.0)
  - `imaginary` (MathML 2.0)
  - `lcm` (MathML 2.0)
- relations
  - `eq`
  - `neq`
  - `gt`

- lt
- geq
- leq
- equivalent (MathML 2.0)
- approx (MathML 2.0)

- calculus and vector calculus
  - int
  - diff
  - partialdiff
  - lowlimit
  - uplimit
  - bvar
  - degree
  - divergence (MathML 2.0)
  - grad (MathML 2.0)
  - curl (MathML 2.0)
  - laplacian (MathML 2.0)
- theory of sets
  - set
  - list
  - union
  - intersect
  - in
  - notin
  - subset
  - prsubset
  - notsubset
  - notprsubset
  - setdiff
  - card (MathML 2.0)
- sequences and series
  - sum
  - product
  - limit
  - tendsto
- elementary classical functions
  - exp
  - ln
  - log
  - sin
  - cos
  - tan
  - sec
  - csc
  - cot
  - sinh
  - cosh

  - `tanh`
  - `sech`
  - `csch`
  - `coth`
  - `arcsin`
  - `arccos`
  - `arctan`
  - `arccosh`
  - `arccot`
  - `arccoth`
  - `arccsc`
  - `arccsch`
  - `arcsec`
  - `arcsech`
  - `arcsinh`
  - `arctanh`
- statistics
  - `mean`
  - `sdev`
  - `variance`
  - `median`
  - `mode`
  - `moment`
- linear algebra
  - `vector`
  - `matrix`
  - `matrixrow`
  - `determinant`
  - `transpose`
  - `selector`
  - `vectorproduct` (MathML 2.0)
  - `scalarproduct` (MathML 2.0)
  - `outerproduct` (MathML 2.0)
- semantic mapping elements
  - `annotation`
  - `semantics`
  - `annotation-xml`
- constant and symbol elements
  - `integers` (MathML2.0)
  - `reals` (MathML2.0)
  - `rationals` (MathML2.0)
  - `naturalnumbers` (MathML2.0)
  - `complexes` (MathML2.0)
  - `primes` (MathML2.0)
  - `exponentiale` (MathML2.0)
  - `imaginaryi` (MathML2.0)
  - `notanumber` (MathML2.0)

- true (MathML2.0)
- false (MathML2.0)
- emptyset (MathML2.0)
- pi (MathML2.0)
- eulergamma (MathML2.0)
- infinity (MathML2.0)

4.4.1 Token Elements

4.4.1.1 Number (cn)

Discussion

The cn element is used to specify actual numerical constants. The content model must provide sufficient information that a number may be entered as data into a computational system. By default, it represents a signed real number in base 10. Thus, the content normally consists of PCDATA restricted to a sign, a string of decimal digits and possibly a decimal point, or alternatively one of the predefined symbolic constants such as π.

The cn element uses the attribute type to represent other types of numbers such as, for example, integer, rational, real or complex, and uses the attribute base to specify the numerical base.

In addition to simple PCDATA, cn accepts as content PCDATA separated by the (empty) element sep. This determines the different parts needed to construct a rational or complex-cartesian number.

The cn element may also contain arbitrary presentation markup in its content (see Chapter 3) so that its presentation can be very elaborate.

Alternative input notations for numbers are possible, but must be explicitly defined by using the definitionURL and encoding attributes, to refer to a written specification of how a sequence of real numbers separated by <sep/> should be interpreted.

Attributes

All attributes are CDATA:

type Allowed values are real, integer, rational, complex-cartesian, complex-polar, constant
base Number (CDATA for XML DTD) between 2 and 36.
definitionURL URL or URI pointing to an alternative definition.
encoding Syntax of the alternative definition.

Examples

```
<cn type="real"> 12345.7 </cn>
<cn type="integer"> 12345 </cn>
<cn type="integer" base="16"> AB3 </cn>
<cn type="rational"> 12342 <sep/> 2342342 </cn>
<cn type="complex-cartesian"> 12.3 <sep/> 5 </cn>
<cn type="complex-polar"> 2 <sep/> 3.1415 </cn>
<cn type="constant">  &pi; </cn>
```

Default Rendering

By default, a contiguous block of PCDATA contained in a cn element should render as if it were wrapped in an mn presentation element. Similarly, presentation markup contained in a cn element should render as it normally would. A mixture of PCDATA and presentation markup should render as if it were contained wrapped in an mrow element, with contiguous blocks of PCDATAwrapped in mn elements.

However, not all mathematical systems that encounter content based tagging do visual or aural rendering. The receiving applications are free to make use of a number in the manner it normally handles numerical data. Some systems might simplify the rational number 12342/2342342 to 6171/1171171 while pure floating point based systems might approximate this as 0.5269085385e-2. All numbers might be re-expressed in base 10. The role of MathML is simply to record enough information about the mathematical object and its structure so that it may be properly parsed.

The following renderings of the above MathML expressions are included both to help clarify the meaning of the corresponding MathML encoding and as suggestions for authors of rendering applications. In each case, no mathematical evaluation is intended or implied.

- 12345.7
- 12345
- AB3 $_{16}$
- 12342 / 2342342
- 12.3 + 5 i
- Polar(2 , 3.1415)
- π

4.4.1.2 Identifier (ci)

Discussion

The ci element is used to name an identifier in a MathML expression (for example a variable). Such names are used to identify mathematical objects. By default they are assumed to represent complex scalars. The ci element may contain arbitrary presentation markup in its content (see Chapter 3) so that its presentation as a symbol can be very elaborate.

The ci element uses the type attribute to specify the type of object that it represents. Valid types include integer, rational, real, float, complex, constant, and more generally, any of the names of the MathML container elements (e.g. vector) or their type values. The definitionURL and encoding attributes can be used to extend the definition of ci to include other types. For example, a more advanced use might require a complex-vector.

Examples

```
<ci> x </ci>
```

```
<ci type="vector"> V </ci>
```

```
<ci>
  <msub>
    <mi>x</mi>
    <mi>a</mi>
  </msub>
</ci>
```

Default Rendering

If the content of a ci element is tagged using presentation tags, that presentation is used. If no such tagging is supplied then the PCDATA content would typically be rendered as if it were the content of an mi element. A renderer may wish to make use of the value of the type attribute to improve on this. For example, a symbol of type vector might be rendered using a bold face. Typical renderings of the above symbols are:

- x
- V
- x_i

4.4.1.3 Externally defined symbol (csymbol)

Discussion

The csymbol element allows a writer to create an element in MathML whose semantics are externally defined (i.e. not in the core MathML content). The element can then be used in a MathML expression as for example an operator or constant. Attributes are used to give the syntax and location of the external definition of the symbol semantics.

Use of csymbol for referencing external semantics can be contrasted with use of the semantics to attach additional information in-line (ie. within the MathML fragment) to a MathML construct. See Section 4.2.6

Attributes

All attributes are CDATA:

definitionURL Pointer to external definition of the semantics of the symbol. MathML does not specify a particular syntax in which this definition should be written.
encoding Gives the syntax of the definition pointed to by definitionURL. An application can then test the value of this attribute to determine whether it is able to process the target of the definitionURL. This syntax might be text, or a formal syntax such as OpenMath.

Examples

```
<!-- reference to OpenMath formal syntax definition of Bessel function -->
<apply>
  <csymbol encoding="OpenMath"
           definitionURL="www.openmath.org/cds/BesselFunctions.ocd">
    <msub><mi>J</mi><mn>0</mn></msub>
  </csymbol>
  <ci>y</ci>
</apply>

<!-- reference to human readable text description of Boltzmann's constant -->
<csymbol encoding="text"
         definitionURL="www.uni.edu/universalconstants/Boltzmann.htm">
  k
</csymbol>
```

Default Rendering

By default, a contiguous block of PCDATA contained in a csymbol element should render as if it were wrapped in an mo presentation element. Similarly, presentation markup contained in a csymbol element should render as it normally would. A mixture of PCDATA and presentation markup should render as if it were contained wrapped in an mrowelement, with contiguous blocks of PCDATA wrapped in mo elements. The examples above would render by default as

- $J_0(y)$
- k

As csymbol is used to support reference to externally defined semantics, it is a MathML error to have embedded content MathML elements within the csymbolelement.

4.4.2 Basic Content Elements

4.4.2.1 Apply (apply)

Discussion

The apply element allows a function or operator to be applied to its arguments. Nearly all expression construction in MathML content markup is carried out by applying operators or functions to arguments. The first child of apply is the operator, to be applied, with the other child elements as arguments.

The apply element is conceptually necessary in order to distinguish between a function or operator, and an instance of its use. The expression constructed by applying a function to 0 or more arguments is always an element from the range of the function.

Proper usage depends on the operator that is being applied. For example, the plus operator may have zero or more arguments. while the minus operator requires one or two arguments to be properly formed.

If the object being applied as a function is not already one of the elements known to be a function (such as fn, sin or plus) then it is treated as if it were the contents of an fn element.

Some operators such as diff and int make use of 'named' arguments. These special arguments are elements that appear as children of the apply element and identify 'parameters'such as the variable of differentiation or the domain of integration. These elements are discussed further in Section 4.2.3.2.

Examples

```
<apply>
  <factorial/>
  <cn>3</cn>
</apply>
```

```
<apply>
  <plus/>
  <cn>3</cn>
  <cn>4</cn>
</apply>
```

```
<apply>
  <sin/>
  <ci>x</ci>
</apply>
```

Default Rendering

A mathematical system that has been passed an apply element is free to do with it whatever it normally does with such mathematical data. It may be that no rendering is involved (e.g. a syntax validator), or that the 'function application' is evaluated and that only the result is rendered (e.g. sin(0) $\rightarrow$ 0).

When an unevaluated 'function application' is rendered there are a wide variety of appropriate renderings. The choice often depends on the function or operator being applied. Applications of basic operations such as plus are generally presented using an infix notation while applications of sinwould use a more traditional functional notation such as sin(x). Consult the default rendering for the operator being applied.

Applications of user-defined functions (see csymbol, fn) that are not evaluated by the receiving or rendering application would typically render using a traditional functional notation unless an alternative presentation is specified using the semantics tag.

4.4.2.2 Relation (reln)

Discussion

The reln element was used in MathML 1.0 to construct an equation or relation. Relations were constructed in a manner exactly analogous to the use of apply. This usage is deprecated in MathML 2.0 in favour of the more generally usable apply.

The first child of reln is the relational operator, to be applied, with the other child elements acting as arguments. See Section 4.2.4 for further details.

Examples

```
<reln>
  <eq/>
  <ci> a </ci>
  <ci> b </ci>
</reln>
```

```
<reln>
  <lt/>
  <ci> a </ci>
  <ci> b </ci>
</reln>
```

Default Rendering

- $a = b$
- $a < b$

4.4.2.3 Function (fn)

Discussion

The fn element makes explicit the fact that a more general (possibly constructed) MathML object is being used in the same manner as if it were a pre-defined function such as sin or plus.

fn has exactly one child element, used to give the name (or presentation form) of the function. When fn is used as the first child of an apply, the number of following arguments is determined by the contents of the fn.

In MathML 1.0, fn was also the primary mechanism used to extend the collection of 'known' mathematical functions. This usage is now deprecated in favour of the more generally applicable csymbol element. (New functions may also be introduced by using declare in conjunction with a lambda expression.)

Examples

```
<fn><ci> L </ci> </fn>
```

```
<apply>
  <fn>
  <apply>
    <plus/>
    <ci> f </ci>
    <ci> g </ci>
  </apply>
  </fn>
  <ci>z</ci>
</apply>
```

Default Rendering

An `fn` object is rendered in the same way as its content. A rendering application may add additional adornments such as parentheses to clarify the meaning.

- L
- $(f+g)z$

4.4.2.4 Interval (`interval`)

Discussion

The `interval` element is used to represent simple mathematical intervals of the real number line. It takes an attribute `closure`, which can take on any of the values `open`, `closed`, `open-closed`, or `closed-open`, with a default value of `closed`.

More general domains are constructed by using the `condition` and `bvar` elements to bind free variables to constraints.

The `interval` element expects *either* two child elements that evaluate to real numbers *or* one child element that is a `condition` defining the `interval`.

Examples

```
<interval>
  <ci> a </ci>
  <ci> b </ci>
</interval>
```

```
<interval closure="open-closed">
  <ci> a </ci>
  <ci> b </ci>
</interval>
```

Default Rendering

- $[a,b]$
- $(a,b]$

4.4.2.5 Inverse (inverse)

Discussion

The inverse element is applied to a function in order to construct a generic expression for the functional inverse of that function. (See also the discussion of inverse in Section 4.2.1.5). As with other MathML functions, inverse may either be applied to arguments, or it may appear alone, in which case it represents an abstract inversion operator acting on other functions.

A typical use of the inverse element is in an HTML document discussing a number of alternative definitions for a particular function so that there is a need to write and define $f^{(-1)}$(x). To associate a particular definition with $f^{(-1)}$, use the definitionURL and encodingattributes.

Examples

```
<apply>
  <inverse/>
  <ci> f </ci>
</apply>
```

```
<apply>
  <inverse definitionURL="../MyDefinition.htm" encoding="text"/>
  <ci> f </ci>
</apply>
```

```
<apply>
  <apply><inverse/>
    <ci type="matrix"> a </ci>
  </apply>
  <ci> A </ci>
</apply>
```

Default Rendering

The default rendering for a functional inverse makes use of a parenthesized exponent as in $f^{(-1)}(x)$.

4.4.2.6 Separator (sep)

Discussion

The sep element is to separate PCDATA into separate tokens for parsing the contents of the various specialized forms of the cn elements. For example, sep is used when specifying the real and imaginary parts of a complex number (see Section 4.4.1). If it occurs between MathML elements, it is a MathML error.

Examples

```
<cn type="complex"> 3 <sep/> 4 </cn>
```

Default Rendering

The sep element is not directly rendered (see Section 4.4.1).

4.4.2.7 Condition (`condition`)

Discussion

The `condition` element is used to place a condition on one or more free variables or identifiers. The conditions may be specified in terms of relations that are to be satisfied by the variables, including general relationships such as set membership.

It is used to define general sets and lists in situations where the elements cannot be explicitly enumerated. Condition contains either a single `apply` or `reln`element; the `apply` element is used to construct compound conditions. For example, it is used below to describe the set of all x such that $x < 5$. See the discussion on sets in Section 4.4.6. See Section 4.2.5 for further details.

Examples

```
<condition>
  <apply><in/><ci> x </ci><ci type="set"> R </ci></apply>
</condition>
```

```
<condition>
  <apply>
    <and/>
    <apply><gt/><ci> x </ci><cn> 0 </cn></apply>
    <apply><lt/><ci> x </ci><cn> 1 </cn></apply>
  </apply>
</condition>
```

```
<apply>
  <max/>
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply> <and/>
      <apply><gt/><ci> x </ci><cn> 0 </cn></apply>
      <apply><lt/><ci> x </ci><cn> 1 </cn></apply>
    </apply>
  </condition>
  <apply>
    <minus/>
    <ci> x </ci>
    <apply>
      <sin/>
      <ci> x </ci>
    </apply>
  </apply>
</apply>
```

Default Rendering

- $x \in \mathbb{R}$
- $x > 0 \land x < 1$
- $\max_x \{ x - \sin x \mid 0 < x < 1 \}$

4.4.2.8 Declare (`declare`)

Discussion

The `declare` construct has two primary roles. The first is to change or set the default attribute values for a specific mathematical object. The second is to establish an association between a ‘name’ and an object. Once a declaration is in effect, the ‘name’ object acquires the new attribute settings, and (if the second object is present) all the properties of the associated object.

The various attributes of the `declare` element assign properties to the object being declared or determine where the declaration is in effect.

By default, the scope of a declaration is ‘local’ to the surrounding container element. Setting the value of the `scope` attribute to `global` extends the scope of the declaration to the enclosing `math`element. As discussed in Section 4.3.2.8, MathML contains no provision for making document-wide declarations at present, though it is anticipated that this capability will be added in future revisions of MathML, when supporting technologies become available. `declare` takes one or two children. The first child, which is mandatory, is a `ci` containing the identifier being declared:

```
<declare type="vector"> <ci> V </ci> </declare>
```

The second child, which is optional, is a constructor initialising the variable:

```
<declare type="vector">
  <ci> V </ci>
  <vector>
    <cn> 1 </cn><cn> 2 </cn><cn> 3 </cn>
  </vector>
</declare>
```

The constructor type and the type of the element declared must agree. For example, if the type attribute of the declaration is `fn`, the second child (constructor) must be an element equivalent to an `fn` element (This would include actual `fn` elements, `lambda` elements and any of the defined function in the basic set of content tags.) If no type is specified in the declaration then the type attribute of the declared name is set to the type of the constructor (second child) of the declaration. The type attribute of the declaration can be especially useful in the special case of the second element being a semantic tag.

Attributes

All attributes are `CDATA`:

`type` defines the MathML element type of the identifier declared.
`scope` defines the scope of application of the declaration.
`nargs` number of arguments for function declarations.
`occurrence` describes operator usage as `prefix`, `infix` or `function-model`indications.
`definitionURL` URI pointing to detailed semantics of the function.
`encoding` syntax of the detailed semantics of the function.

Examples

The declaration

```
<declare type="fn" nargs="2" scope="local">
  <ci> f </ci>
  <apply>
    <plus/>
    <ci> F </ci><ci> G </ci>
  </apply>
</declare>
```

declares f to be a two-variable function with the property that $f(x, y) = (F+G)(x, y)$.

The declaration

```
<declare type="fn">
  <ci> J </ci>
  <lambda>
    <bvar><ci> x </ci></bvar>
    <apply><ln/>
      <ci> x </ci>
    </apply>
  </lambda>
</declare>
```

associates the name J with a one-variable function defined so that $J(x) = \ln y$. (Note that because of the type attribute of the `declare` element, the second argument must be something of type `fn`, namely a known function like `sin`, an `fn` construct, or a `lambda`construct.)

The `type` attribute on the declaration is only necessary if if the type cannot be inferred from the type of the second argument.

Even when a declaration is in effect it is still possible to override attributes values selectively as in `<ci type="integer"> V </ci>`. This capability is needed in order to write statements of the form 'Let S be a member of S'.

Default Rendering

Since the `declare` construct is not directly rendered, most declarations are likely to be invisible to a reader. However, declarations can produce quite different effects in an application which evaluates or manipulates MathML content. While the declaration

```
<declare>
  <ci> v </ci>
  <vector>
    <cn> 1 </cn>
    <cn> 2 </cn>
    <cn> 3 </cn>
  </vector>
</declare>
```

is active the symbol v acquires all the properties of the vector, and even its dimension and components have meaningful values. This may affect how v is rendered by some applications, as well as how it is treated mathematically.

4.4.2.9 Lambda (`lambda`)

Discussion

The `lambda` element is used to construct a user-defined function from an expression and one or more free variables. The lambda construct with n internal variables takes n+1 children. The first n children identify the variables that are used as placeholders in the last child for actual parameter values. See Section 4.2.2.2 for further details.

Examples

The first example presents a simple lambda construct.

```
<lambda>
  <bvar><ci> x </ci></bvar>
  <apply><sin/>
    <apply>
      <plus/>
      <ci> x </ci>
      <cn> 1 </cn>
    </apply>
  </apply>
</lambda>
```

The next example constructs a one-argument function in which the argument b specifies the upper bound of a specific definite integral.

```
<lambda>
  <bvar><ci> b </ci></bvar>
  <apply>
    <int/>
    <bvar>
       <ci> x </ci>
    </bvar>
    <lowlimit>
       <ci> a </ci>
    </lowlimit>
    <uplimit>
       <ci> b </ci>
    </uplimit>
    <apply><fn><ci> f </ci></fn>
       <ci> x </ci>
    </apply>
  </apply>
</lambda>
```

Such constructs are often used in conjunction with `declare` to construct new functions.

Default Rendering

- $\lambda(x, \sin x + 1)$
- $\lambda(b, \int_a^b f(x)\,\mathrm{d}x)$

4.4.2.10 Function composition (`compose`)

Discussion

The `compose` element represents the function composition operator. Note that MathML makes no assumption about the domain and range of the constituent functions in a composition; the domain of the resulting composition may be empty.

To override the default semantics for the `compose` element, or to associate a more specific definition for function composition, use the `definitionURL` and `encoding`attributes. See Section 4.2.3 for further details.

Examples

```
<apply>
  <compose/>
  <fn><ci> f </ci></fn>
  <fn><ci> g </ci></fn>
</apply>
```

```
<apply>
  <compose/>
  <ci type="fn"> f </ci>
  <ci type="fn"> g </ci>
  <ci type="fn"> h </ci>
</apply>
```

```
<apply>
  <apply><compose/>
    <fn><ci> f </ci></fn>
    <fn><ci> g </ci></fn>
  </apply>
  <ci> x </ci>
</apply>
```

```
<apply>
  <fn><ci> f </ci></fn>
  <apply>
    <fn><ci> g </ci></fn>
    <ci> x </ci>
  </apply>
</apply>
```

Default Rendering

- $f \circ g$
- $f \circ g \circ h$
- $(f \circ g)(x)$
- $f(g(x))$

4.4.2.11 Identity function (`ident`)

Discussion

The `ident` element represents the identity function. MathML makes no assumption about the function space in which the identity function resides. That is, proper interpretation of the domain (and hence range) of the identity function depends on the context in which it is used.

To override the default semantics for the `ident`element, or to associate a more specific definition, use the `definitionURL` and `encoding`attributes (see Section 4.2.3).

Examples

```
<apply>
  <eq/>
  <apply><compose/>
    <fn><ci> f </ci></fn>
    <apply><inverse/>
      <fn><ci> f </ci></fn>
    </apply>
  </apply>
  <ident/>
</apply>
```

Default Rendering

$f \circ f^{-1} = \mathrm{id}$

4.4.3 Arithmetic, Algebra and Logic

4.4.3.1 Quotient (`quotient`)

Discussion

The `quotient` element is the operator used for division modulo a particular base. When the `quotient` operator is applied to integer arguments a and b, the result is the 'quotient of $a$divided by b'. That is, `quotient`returns the unique integer, q such that $a = q\ b + r$. (In common usage, q is called the quotient and r is the remainder.)

The `quotient` element takes the attribute `definitionURL` and `encoding`attributes, which can be used override the default semantics.

The `quotient` element is a *binary arithmetic operator* (see Section 4.2.3).

Example

```
<apply>
  <quotient/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

Various mathematical applications will use this data in different ways. Editing applications might choose an image such as shown below, while a computationally based application would evaluate it to 2 when a=13 and b=5.

Default Rendering

There is no commonly used notation for this concept. Some possible renderings are

- quotient of a divided by b
- integer part of a/ b
- $\lfloor a/b \rfloor$

4.4.3.2 Factorial (`factorial`)

Discussion

The `factorial` element is used to construct factorials.

The `factorial` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `factorial` element is a *unary arithmetic operator* (see Section 4.2.3).

Example

```
<apply>
  <factorial/>
  <ci> n </ci>
</apply>
```

If this were evaluated at *n* = 5 it would evaluate to 120.

Default Rendering

$n!$

4.4.3.3 Division (`divide`)

Discussion

The `divide` element is the division operator.

The `divide` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `divide` element is a *binary arithmetic operator* (see Section 4.2.3).

Example

```
<apply>
  <divide/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

As a MathML expression, this does not evaluate. However, on receiving such an expression, some applications may attempt to evaluate and simplify the value. For example, when a=5 and b=2 some mathematical applications may evaluate this to 2.5 while others will treat is as a rational number.

Default Rendering

a/b

4.4.3.4 Maximum and minimum (`max`, `min`)

Discussion

The elements `max` and `min`are used to compare the values of their arguments. They return the maximum and minimum of these values respectively.

The `max` and `min` elements take the `definitionURL` and `encoding` attributes that can be used to override the default semantics.

The `max` and `min` elements are *n-ary arithmetic operators* (see Section 4.2.3).

Examples

When the objects are to be compared explicitly they are listed as arguments to the function as in:

```
<apply>
  <max/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

The elements to be compared may also be described using bound variables with a `condition` element and an expression to be maximised, as in:

```
<apply>
  <min/>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply><notin/><ci> x </ci><ci type="set"> B </ci></apply>
  </condition>
  <apply>
      <power/>
    <ci> x </ci>
    <cn> 2 </cn>
  </apply>
</apply>
```

Note that the bound variable must be stated even if it might be implicit in conventional notation. In MathML1.0, the bound variable and expression to be evaluated (x) could be omitted in the example below: this usage is deprecated in MathML2.0 in favour of explicitly stating the bound variable and expression in all cases:

```
<apply>
  <bvar><ci>x</ci></bvar>
  <max/>
  <condition>
    <apply><and/>
      <apply><in/><ci>x</ci><ci type="set">B</ci></apply>
      <apply><notin/><ci>x</ci><ci type="set">C</ci></apply>
    </apply>
  </condition>
  <ci>x</ci>
</apply>
```

Default Rendering

- $\max\{a,b\}$
- $\min_x\{x^2 \mid x \notin B\}$
- $\max\{x \in B \wedge x \notin C\}$

4.4.3.5 Subtraction (`minus`)

Discussion

The `minus` element is the subtraction operator.

The `minus` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `minus` element can be used as a *unary arithmetic operator* (e.g. to represent - x), or as a *binary arithmetic operator* (e.g. to represent x- y).

Example

```
<apply> <minus/>
  <ci> x </ci>
  <ci> y </ci>
</apply>
```

If this were evaluated at x=5 and y=2 it would yield 3.

Default Rendering

$x - y$

4.4.3.6 Addition (`plus`)

Discussion

The `plus` element is the addition operator.

The `plus` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `plus` element is an *n-ary arithemtic operator* (see Section 4.2.3).

Example

```
<apply>
  <plus/>
  <ci> x </ci>
  <ci> y </ci>
  <ci> z </ci>
</apply>
```

If this were evaluated at x = 5, y = 2 and z = 1 it would yield 8.

Default Rendering

$x + y + z$

4.4.3.7 Exponentiation (`power`)

Discussion

The `power` element is a generic exponentiation operator. That is, when applied to arguments a and b, it returns the value the ' a to the power of b'.

The `power` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `power` element is an *binary arithmetic operator* (see Section 4.2.3).

Example

```
<apply>
  <power/>
  <ci> x </ci>
  <cn> 3 </cn>
</apply>
```

If this were evaluated at x= 5 it would yield 125.

Default Rendering

x^3

4.4.3.8 Remainder (`rem`)

Discussion

The `rem` element is the operator that returns the ‘remainder’ of a division modulo a particular base. When the `rem` operator is applied to integer arguments a and b, the result is the ‘remainder of $a$divided by b’. That is, `rem` returns the unique integer, r such that $a = q\,b + r$, where $r < q$. (In common usage, q is called the quotient and r is the remainder.)

The `rem` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `rem` element is a *binary arithmetic operator* (see Section 4.2.3).

Example

```
<apply>
  <rem/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were evaluated at a = 15 and b = 8 it would yield 7.

Default Rendering

$a \bmod b$

4.4.3.9 Multiplication (`times`)

Discussion

The `times` element is the multiplication operator.

`times` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

Example

```
<apply>
  <times/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were evaluated at $a = 5.5$ and $b = 3$ it would yield 16.5.

Default Rendering

ab

4.4.3.10 Root (root)

Discussion

The root element is used to construct roots. The kind of root to be taken is specified by a degreeelement, which should be given as the first child of the apply element enclosing the root element. Thus, square roots correspond to the case where degree contains the value 2, cube roots correspond to 3, and so on. If no degree is present, a default value of 2 is used.

The root element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The root element is an *operator taking qualifiers* (see Section 4.2.3.2).

Example

The *n*th root of a is is given by

```
<apply>
  <root/>
  <degree><ci type='integer'> n </ci></degree>
  <ci> a </ci>
</apply>
```

Default Rendering

$\sqrt[n]{a}$

4.4.3.11 Greatest common divisor (gcd)

Discussion

The gcd element is used to denote the greatest common divisor of its arguments.

The gcd takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The gcd element is an *n-ary operator* (see Section 4.2.3).

Example

```
<apply> <gcd/>
  <ci> a </ci>
  <ci> b </ci>
  <ci> c </ci>
</apply>
```

If this were evaluated at $a = 15$, $b = 21$, $c = 48$ it would yield 3.

Default Rendering

$\gcd(a,b,c)$

4.4.3.12 And (`and`)

Discussion

The `and` element is the boolean ‘and’ operator.

The `and` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `and` element is an *n-ary logical operator* (see Section 4.2.3).

Example

```
<apply>
  <and/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were evaluated and both a and b had truth values of `true`, then the value would be `true`.

Default Rendering

$$a \wedge b$$

4.4.3.13 Or (`or`)

Discussion

The `or` element is the boolean ‘or’ operator.

The `or` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `or` element is an *n-ary logical operator* (see Section 4.2.3).

Example

```
<apply>
  <or/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

Default Rendering

$a \vee b$

4.4.3.14 Exclusive Or (`xor`)

Discussion

The `xor` element is the boolean 'exclusive or'operator.

`xor` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `xor` element is an *n-ary logical operator* (see Section 4.2.3).

Example

```
<apply>
  <xor/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

Default Rendering

$a \operatorname{xor} b$

4.4.3.15 Not (`not`)

The `not` operator is the boolean 'not' operator.

The `not` element takes the attribute `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `not` element is a *unary logical operator* (see Section 4.2.3).

Example

```
<apply>
  <not/>
  <ci> a </ci>
</apply>
```

Default Rendering

$\neg a$

4.4.3.16 Implies (`implies`)

Discussion

The `implies` element is the boolean relational operator 'implies'.

The `implies` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `implies` element is a *binary logical operator* (see Section 4.2.4).

Example

```
<apply>
  <implies/>
  <ci> A </ci>
  <ci> B </ci>
</apply>
```

Mathematical applications designed for the evaluation of such expressions would evaluate this to `true` when a = `false` and b = `true`.

Default Rendering

$$A \Rightarrow B$$

4.4.3.17 Universal quantifier (`forall`)

The `forall` element represents the universal quantifier of logic. It must used in conjunction with one or more bound variables, an optional `condition` element, and an assertion, which may either take the form of an `apply` or `reln` element.

The `forall` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `forall` element is an *quantifier* (see Section 4.2.3.2).

Examples

The first example encodes a simple identity.

```
<apply>
  <forall/>
  <bvar><ci> x </ci></bvar>
  <apply><eq/>
    <apply>
      <minus/><ci> x </ci><ci> x </ci>
    </apply>
    <cn>0</cn>
  </apply>
</apply>
```

The next example is more involved, and makes use of an optional `condition` element.

```
<apply>
  <forall/>
  <bvar><ci> p </ci></bvar>
  <bvar><ci> q </ci></bvar>
  <condition>
    <apply><and/>
      <apply><in/><ci> p </ci><ci type="set"> Q </ci></apply>
      <apply><in/><ci> q </ci><ci type="set"> Q </ci></apply>
      <apply><lt/><ci> p </ci><ci> q </ci></apply>
    </apply>
  </condition>
  <apply><lt/>
      <ci> p </ci>
      <apply>
          <power/>
        <ci> q </ci>
        <cn> 2 </cn>
      </apply>
  </apply>
</apply>
```

The final example uses both the `forall` and `exists` quantifiers.

```
<apply>
  <forall/>
  <bvar><ci> n </ci></bvar>
  <condition>
    <apply><and/>
      <apply><gt/><ci> n </ci><cn> 0 </cn></apply>
      <apply><in/><ci> n </ci><ci type="set"> Z </ci></apply>
    </apply>
  </condition>
  <apply>
    <exists/>
    <bvar><ci> x </ci></bvar>
    <bvar><ci> y </ci></bvar>
    <bvar><ci> z </ci></bvar>
    <condition>
      <apply><and/>
        <apply><in/><ci> x </ci><ci type="set"> Z </ci></apply>
        <apply><in/><ci> y </ci><ci type="set"> Z </ci></apply>
        <apply><in/><ci> z </ci><ci type="set"> Z </ci></apply>
      </apply>
    </condition>
    <apply>
      <eq/>
      <apply>
        <plus/>
        <apply><power/><ci> x </ci><ci> n </ci></apply>
```

```
        <apply><power/><ci> y </ci><ci> n </ci></apply>
      </apply>
      <apply><power/><ci> z </ci><ci> n </ci></apply>
    </apply>
  </apply>
</apply>
```

Default Rendering

- $\forall x : x - x = 0$
- $\forall p \in \mathbb{Q}, q \in \mathbb{Q}, p < q : p < q^2$
- $\forall n > 0, n \in \mathbb{Z} : \exists x \in \mathbb{Z}, y \in \mathbb{Z}, z \in \mathbb{Z} : x^n + y^n = z^n$

4.4.3.18 Existential quantifier (exists)

The exists element represents the existential quantifier of logic. It must used in conjuction with one or more bound variables, an optional condition element, and an assertion, which may either take the form of an apply or reln element.

The exists element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The exists element is an *quantifier* (see Section 4.2.3.2).

Example

The following example encodes the sense of the expression ‘there exists an x such that $f(x) = 0$’.

```
<apply>
  <exists/>
  <bvar><ci> x </ci></bvar>
  <apply><eq/>
    <apply>
      <fn><ci> f </ci></fn>
      <ci> x </ci>
    </apply>
    <cn>0</cn>
  </apply>
</apply>
```

Default Rendering

$\exists x : f(x) = 0$

4.4.3.19 Absolute Value (abs)

The abs element represents the absolute value of a real quantity or the modulus of a complex quantity.

The abs element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The abs element is a *unary arithmetic operator* (see Section 4.2.3).

Example

The following example encodes the absolute value of x.

```
<apply>
  <abs/>
  <ci> x </ci>
</apply>
```

Default Rendering

$|x|$

4.4.3.20 Complex conjugate (conjugate)

The `conjugate` element represents the complex conjugate of a complex quantity.

The `conjugate` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `conjugate` element is a *unary arithmetic operator* (see Section 4.2.3).

Example

The following example encodes the conjugate of x + i y.

```
<apply>
  <conjugate/>
  <apply>
    <plus/>
    <ci> x </ci>
    <apply><times/>
      <cn> &ImaginaryI; </cn>
      <ci> y </ci>
    </apply>
  </apply>
</apply>
```

Default Rendering

$$\overline{x + \mathrm{i} y}$$

4.4.3.21 Argument (arg)

The `arg` operator (introduced in MathML 2.0)) gives the ‘argument’ of a complex number, which is the angle (in radians) it makes with the positive real axis. Real negative numbers have argument equal to + π.

The `arg` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `arg` element is a *unary arithmetic operator* (see Section 4.2.3).

Example

The following example encodes the argument operation on $x + \mathrm{i}\, y$.

```
<apply>
  <arg/>
  <apply><plus/>
    <ci> x </ci>
    <apply><times/>
      <cn> &ImaginaryI; </cn>
      <ci> y </ci>
    </apply>
  </apply>
</apply>
```

Default Rendering

$$\arg(x+\mathrm{i}y)$$

4.4.3.22 Real part (real)

The `real` operator (introduced in MathML 2.0) gives the real part of a complex number, that is the x component in $x + \mathrm{i}\, y$

The `real` element takes the attributes `encoding, definitionURL` that can be used to override the default semantics.

The `real` element is a *unary arithmetic operator* (see Section 4.2.3).

Example

The following example encodes the real operation on $x + \mathrm{i}\, y$.

```
<apply>
  <real/>
  <apply><plus/>
    <ci> x </ci>
    <apply><times/>
      <cn> &ImaginaryI; </cn>
      <ci> y </ci>
    </apply>
  </apply>
</apply>
```

A MathML-aware evaluation system would return the x component, suitably encoded.

Default Rendering

$$\Re(x+\mathrm{i}y)$$

4.4.3.23 Imaginary part (imaginary)

The `imaginary` operator (introduced in MathML 2.0) gives the imaginary part of a complex number, that is the y component in $x + \mathrm{i}\, y$

The `imaginary` element takes the attributes `encoding, definitionURL` that can be used to override the default semantics.

The `imaginary` element is a *unary arithmetic operator* (see Section 4.2.3).

Example

The following example encodes the imaginary operation on $x + \mathrm{i}\, y$.

```
<apply>
  <imaginary/>
  <apply><plus/>
    <ci> x </ci>
    <apply><times/>
      <cn> &ImaginaryI; </cn>
      <ci> y </ci>
    </apply>
  </apply>
</apply>
```

A MathML-aware evaluation system would return the y component, suitably encoded.

Default Rendering

$$\Im(x+\mathrm{i}y)$$

4.4.3.24 Lowest common multiple (lcm)

Discussion

The `lcm` element (introduced in MathML 2.0) is used to denote the lowest common multiple of its arguments.

The `lcm` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `lcm` element is an *n-ary operator* (see Section 4.2.3).

Example

```
<apply> <lcm/>
  <ci> a </ci>
  <ci> b </ci>
  <ci> c </ci>
</apply>
```

If this were evaluated at $a = 2$, $b = 4$, $c = 6$ it would yield 12.

Default Rendering

$$\mathrm{lcm}(a,b,c)$$

4.4.4 Relations

4.4.4.1 Equals (eq)

Discussion

The `eq` element is the relational operator ‘equals’.

The `eq` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `equals` element is an *n-ary relation* (see Section 4.2.3.2).

Example

```
<apply>
  <eq/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were tested at a = 5.5 and b = 6 it would yield the truth value `false`.

Default Rendering

$a = b$

4.4.4.2 Not Equals (neq)

Discussion

The `neq` element is the ‘not equal to’ relational operator.

`neq` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `neq` element is an *binary relation* (see Section 4.2.4).

Example

```
<apply>
  <neq/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were tested at a = 5.5 and b = 6 it would yield the truth value `true`.

Default Rendering

$$a \neq b$$

4.4.4.3 Greater than (gt)

Discussion

The `gt` element is the ‘greater than’ relational operator.

The `gt` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `gt` element is an *n-ary relation* (see Section 4.2.4).

Example

```
<apply>
  <gt/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were tested at a = 5.5 and b = 6 it would yield the truth value `false`.

Default Rendering

$a > b$

4.4.4.4 Less Than (`lt`)

Discussion

The `lt` element is the ‘less than’ relational operator.

The `lt` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `lt` element is an *n-ary relation* (see Section 4.2.4).

Example

```
<apply>
  <lt/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were tested at a = 5.5 and b = 6 it would yield the truth value ‘true’.

Default Rendering

$a < b$

4.4.4.5 Greater Than or Equal (`geq`)

Discussion

The `geq` element is the relational operator ‘greater than or equal’.

The `geq` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `geq` element is an *n-ary relation* (see Section 4.2.4).

Example

```
<apply>
  <geq/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If this were tested for $a = 5.5$ and $b = 5.5$ it would yield the truth value `true`.

Default Rendering

$$a \geq b$$

4.4.4.6 Less Than or Equal (`leq`)

Discussion

The `leq` element is the relational operator ‘less than or equal’.

The `leq` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `leq` element is an *n-ary relation* (see Section 4.2.4).

Example

```
<apply>
  <leq/>
  <ci> a </ci>
  <ci> b </ci>
</apply>
```

If $a = 5.4$ and $b = 5.5$ this will yield the truth value `true`.

Default Rendering

$$a \leq b$$

4.4.4.7 Equivalent (`equivalent`)

Discussion

The `equivalent` element is the ‘equivalence’ relational operator.

The `equivalent` element takes the attributes `encoding, definitionURL` that can be used to override the default semantics.

The `equivalent` element is an *n-ary relation* (see Section 4.2.3.2).

Example

```
<apply>
  <equivalent/>
  <ci> a </ci>
  <apply>
    <not/>
    <apply> <not/> <ci> a </ci> </apply>
  </apply>
</apply>
```

This yields the truth value `true` for all values of a.

Default Rendering

$$a \equiv \neg(\neg a)$$

4.4.4.8 Approximately (`approx`)

Discussion

The `approx` element is the relational operator ‘approximately equal’.

The `approx` element takes the attributes `encoding, definitionURL` that can be used to override the default semantics.

The `approx` element is a *binary relation* (see Section 4.2.3.2).

Example

```
<apply>
  <approx/>
  <cn type="rational"> 22 <sep/> 7 </cn>
  <cn type="constant"> &pi; </cn>
</apply>
```

Default Rendering

$$a \approx b$$

4.4.5 Calculus and Vector Calculus

4.4.5.1 Integral (`int`)

Discussion

The `int` element is the operator element for an integral. The lower limit, upper limit and bound variable are given by (optional) child elements, `lowlimit`, `uplimit` and `bvar` in the enclosing `apply` element. The integrand is also specified as a child element of the enclosing `apply` element.

The domain of integration may alternatively be specified by using an `interval` element, or by a `condition` element. In such cases, if a bound variable of integration is intended, it must be specified explicitly. (The condition may involve more than one symbol.)

The `int` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `int` element is an *operator taking qualifiers* (see Section 4.2.3.2).

Examples

This example specifies a `lowlimit`, `uplimit`, and `bvar`.

```
<apply>
  <int/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <lowlimit>
    <cn> 0 </cn>
  </lowlimit>
  <uplimit>
    <ci> a </ci>
  </uplimit>
  <apply>
    <fn><ci> f </ci></fn>
    <ci> x </ci>
  </apply>
</apply>
```

This example specifies the domain of integration with an `interval` element.

```
<apply>
  <int/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <interval>
    <ci> a </ci>
    <ci> b </ci>
  </interval>
  <apply><cos/>
    <ci> x </ci>
  </apply>
</apply>
```

The final example specifies the domain of integration with an `condition` element.

```
<apply>
  <int/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <condition>
    <apply><in/>
      <ci> x </ci>
      <ci type="set"> D </ci>
    </apply>
  </condition>
  <apply><fn><ci> f </ci></fn>
    <ci> x </ci>
  </apply>
</apply>
```

Default Rendering

$$\int_0^a f(x)\,\mathrm{d}x$$

$$\int_a^b \cos x\,\mathrm{d}x$$

$$\int_{x\in D} f(x)\,\mathrm{d}x$$

4.4.5.2 Differentiation (diff)

Discussion

The diff element is the differentiation operator element for functions of a single real variable. The bound variable is given by a bvar element that is a child of the containing apply element. The bvar elements may also contain a degree element, which specifies the order of the derivative to be taken.

The diff element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The diff element is an *operator taking qualifiers* (see Section 4.2.3.2).

Example

```
<apply>
  <diff/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <apply><fn><ci> f </ci></fn>
    <ci> x </ci>
  </apply>
</apply>
```

Default Rendering

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

4.4.5.3 Partial Differentiation (partialdiff)

Discussion

The `partialdiff` element is the partial differentiation operator element for functions of several real variables. The bound variables are given by `bvar` elements, which are children of the containing `apply` element. The `bvar`elements may also contain a `degree` element, which specifies the order of the partial derivative to be taken in that variable.

The `partialdiff` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `partialdiff` element is an *operator taking qualifiers* (see Section 4.2.3.2).

Example

```
<apply>
  <partialdiff/>
  <bvar>
     <ci> x </ci>
     <degree>
        <cn> 2 </cn>
     </degree>
  </bvar>
  <bvar>
     <ci> y </ci>
  </bvar>
  <apply><fn><ci> f </ci></fn>
    <ci> x </ci>
    <ci> y </ci>
  </apply>
</apply>
```

Default Rendering

$$\frac{\partial^3}{\partial x^2 \partial y} f(x,y)$$

4.4.5.4 Lower limit (lowlimit)

Discussion

The `lowlimit` element is the container element used to indicate the ‘lower limit’ of an operator using qualifiers. For example, in an integral, it can be used to specify the lower limit of integration. Similarly, it is also used to specify the lower limit of an index for sums and products.

The meaning of the `lowlimit` element depends on the context it is being used in. For further details about how *qualifiers* are used in conjunction with operators taking qualifiers, consult Section 4.2.3.2.

Example

```
<apply>
  <int/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <lowlimit>
    <ci> a </ci>
  </lowlimit>
  <uplimit>
    <ci> b </ci>
  </uplimit>
  <apply><fn><ci> f </ci></fn>
     <ci> x </ci>
  </apply>
</apply>
```

Default Rendering

The default rendering of the `lowlimit` element and its contents depends on the context. In the preceding example, it should be rendered as a subscript to the integral sign:

$$\int_a^b f(x)\,\mathrm{d}x$$

Consult the descriptions of individual operators that make use of the `lowlimit` construct for default renderings.

4.4.5.5 Upper limit (`uplimit`)

Discussion

The `uplimit` element is the container element used to indicate the ‘upper limit’ of an operator using qualifiers. For example, in an integral, it can be used to specify the upper limit of integration. Similarly, it is also used to specify the upper limit of an index for sums and products.

The meaning of the `uplimit` element depends on the context it is being used in. For further details about how *qualifiers* are used in conjunction with operators taking qualifiers, consult Section 4.2.3.2.

Example

```
<apply>
  <int/>
  <bvar>
     <ci> x </ci>
  </bvar>
  <lowlimit>
     <ci> a </ci>
  </lowlimit>
  <uplimit>
     <ci> b </ci>
  </uplimit>
  <apply><fn><ci> f </ci></fn>
     <ci> x </ci>
  </apply>
</apply>
```

Default Rendering

The default rendering of the `uplimit` element and its contents depends on the context. In the preceding example, it should be rendered as a superscript to the integral sign:

$$\int_a^b f(x)\,\mathrm{d}x$$

Consult the descriptions of individual operators that make use of the `uplimit` construct for default renderings.

4.4.5.6 Bound variable (`bvar`)

Discussion

The `bvar` element is the container element for the 'bound variable' of an operation. For example, in an integral it specifies the variable of integration. In a derivative, it indicates which variable with respect to which a function is being differentiated. When the `bvar` element is used to quantify a derivative, the `bvar` element may contain a child `degree` element that specifies the order of the derivative with respect to that variable. The `bvar`element is also used for the internal variable in sums and products and for the bound variable used with the universal and existential quantifiers `forall` and `exists`.

The meaning of the `bvar` element depends on the context it is being used in. For further details about how *qualifiers* are used in conjunction with operators taking qualifiers, consult Section 4.2.3.2.

Examples

```
<apply>
  <diff/>
  <bvar>
    <ci> x </ci>
    <degree>
      <cn> 2 </cn>
    </degree>
  </bvar>
  <apply>
    <power/>
    <ci> x </ci>
    <cn> 4 </cn>
  </apply>
</apply>
```

```
<apply>
  <int/>
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply><in/><ci> x </ci><ci> D </ci></apply>
  </condition>
  <apply><fn><ci> f </ci></fn>
    <ci> x </ci>
  </apply>
</apply>
```

Default Rendering

The default rendering of the `bvar` element and its contents depends on the context. In the preceding examples, it should be rendered as the x in the d x of the integral, and as the x in the denominator of the derivative symbol:

$$\frac{\mathrm{d}x^4}{\mathrm{d}x^2}$$

$$\int_{x\in D} f(x)\,\mathrm{d}x$$

Note that in the case of the derivative, the default rendering of the `degree` child of the `bvar` element is as an exponent.

Consult the descriptions of individual operators that make use of the `bvar` construct for default renderings.

4.4.5.7 Degree (`degree`)

Discussion

The `degree` element is the container element for the ‘degree’ or ‘order’ of an operation. There are a number basic mathematical constructs that come in families, such as derivatives and moments. Rather than introduce special elements for each of these families, MathML uses a single general construct, the `degree` element for this concept of ‘order’.

The meaning of the `degree` element depends on the context it is being used in. For further details about how *qualifiers* are used in conjunction with operators taking qualifiers, consult Section 4.2.3.2.

Example

```
<apply>
  <partialdiff/>
  <bvar>
    <ci> x </ci>
    <degree>
      <ci> n </ci>
    </degree>
  </bvar>
  <bvar>
    <ci> y </ci>
    <degree>
      <ci> m </ci>
    </degree>
  </bvar>
  <apply><sin/>
    <apply> <times/>
      <ci> x </ci>
      <ci> y </ci>
    </apply>
  </apply>
</apply>
```

Default Rendering

The default rendering of the `degree` element and its contents depends on the context. In the preceding example, the `degree` elements would be rendered as the exponents in the differentiation symbols:

$$\frac{\partial^{n+m}}{\partial x^n \partial y^m} \sin(xy)$$

Consult the descriptions of individual operators that make use of the `degree` construct for default renderings.

4.4.5.8 Divergence (`divergence`)

Discussion

The `divergence` element is the vector calculus divergence operator, often called div.

The `divergence` element takes the attributes `encoding, definitionURL` that can be used to override the default semantics.

The `divergence` element is an *unary calculus operator* (see Section 4.2.3).

Example

```
<apply>
  <divergence/>
  <ci> a </ci>
</apply>
```

If a is a vector field defined inside a closed surface S enclosing a volume V, then the divergence of a is given by

```
<apply>
  <limit/>
  <bvar>
    <ci> V </ci>
  </bvar>
  <condition>
    <apply>
      <tendsto/>
      <ci> V </ci>
      <cn> 0 </cn>
    </apply>
  </condition>
  <apply>
    <divide/>
    <apply><int encoding="text" definitionURL="SurfaceIntegrals.htm"/>
      <bvar>
        <ci> S</ci>
      </bvar>
    <ci> a </ci>
    </apply>
    <ci> V </ci>
  </apply>
</apply>
```

Default Rendering

$\operatorname{div} a$

4.4.5.9 Gradient (`grad`)

Discussion

The `grad` element is the vector calculus gradient operator, often called grad.

The `grad` element takes the attributes `encoding, definitionURL` that can be used to override the default semantics.

The `grad` element is an *unary calculus operator* (see Section 4.2.3).

Example

```
<apply>
  <grad/>
  <ci> f</ci>
</apply>
```

Where for example f is a scalar function and $f(x, y, z) = k$ defines a surface S

Default Rendering

$\operatorname{grad} f$

4.4.5.10 Curl (curl)

Discussion

The curl element is the vector calculus curl operator.

The curl element takes the attributes encoding, definitionURL that can be used to override the default semantics.

The curl element is an *unary calculus operator* (see Section 4.2.3).

Example

```
<apply>
  <curl/>
  <ci> a </ci>
</apply>
```

Where for example a is a vector.

Default Rendering

$$\mathrm{curl}\, a$$

4.4.5.11 Laplacian (laplacian)

Discussion

The laplacian element is the vector calculus laplacian operator.

The laplacian element takes the attributes encoding, definitionURL that can be used to override the default semantics.

The laplacian element is an *unary calculus operator* (see Section 4.2.3).

Example

```
<apply>
  <eq/>
  <apply><laplacian/>
    <ci> f </ci>
  </apply>
  <apply>
    <divergence/>
    <apply><grad/>
      <ci> f </ci>
    </apply>
  </apply>
</apply>
```

Where for example f is a vector

Default Rendering

$$\nabla^2 f$$

4.4.6 Theory of Sets

4.4.6.1 Set (set)

Discussion

The set element is the container element that constructs a set of elements. The elements of a set can be defined either by explicitly listing the elements, or by using the bvar and condition elements.

The set element is a *constructor element* (see Section 4.2.2.2).

Examples

```
<set>
  <ci> b </ci>
  <ci> a </ci>
  <ci> c </ci>
</set>
```

```
<set>
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply><lt/>
      <ci> x </ci>
      <cn> 5 </cn>
    </apply>
  </condition>
</set>
```

Default Rendering

- $\{a,b,c\}$
- $\{x \mid x < 5\}$

4.4.6.2 List (list)

Discussion

The list element is the container element that constructs a list of elements. Elements can be defined either by explicitly listing the elements, or by using the bvar and condition elements.

Lists differ from sets in that there is an explicit order to the elements. Two orders are supported: lexicographic and numeric. The kind of ordering that should be used is specified by the order attribute.

The list element is a *constructor element* (see Section 4.2.2.2).

Examples

```
<list>
  <ci> a </ci>
  <ci> b </ci>
  <ci> c </ci>
</list>
```

```
<list order="numeric">
  <bvar><ci> x </ci></bvar>
  <condition>
    <apply><lt/>
      <ci> x </ci>
      <cn> 5 </cn>
    </apply>
  </condition>
</list>
```

Default Rendering

- $[a,b,c]$
- $[x \mid x < 5]$

4.4.6.3 Union (`union`)

Discussion

The `union` element is the operator for a set-theoretic union or join of two (or more) sets.

The `union` attribute takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `union` element is an *n-ary set operator* (see Section 4.2.3).

Example

```
<apply>
  <union/>
  <ci> A </ci>
  <ci> B </ci>
</apply>
```

Default Rendering

$$A \cup B$$

4.4.6.4 Intersect (`intersect`)

Discussion

The `intersect` element is the operator for the set-theoretic intersection or meet of two (or more) sets.

The `intersect` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `intersect` element is an *n-ary set operator* (see Section 4.2.3).

Example

```
<apply>
  <intersect/>
  <ci type="set"> A </ci>
  <ci type="set"> B </ci>
</apply>
```

Default Rendering

$$A \cap B$$

4.4.6.5 Set inclusion (in)

Discussion

The in element is the relational operator used for a set-theoretic inclusion (‘is in’ or ‘is a member of’).

The in element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The in element is a *binary set relation* (see Section 4.2.4).

Example

```
<apply>
  <in/>
  <ci> a </ci>
  <ci type="set"> A </ci>
</apply>
```

Default Rendering

$$a \in A$$

4.4.6.6 Set exclusion (notin)

Discussion

The notin element is the relational operator element used for set-theoretic exclusion (‘is not in’ or ‘is not a member of’).

The notin element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The notin element is a *binary set relation* (see Section 4.2.4).

Example

```
<apply>
  <notin/>
  <ci> a </ci>
  <ci> A </ci>
</apply>
```

Default Rendering

$a \notin A$

4.4.6.7 Subset (`subset`)

Discussion

The `subset` element is the relational operator element for a set-theoretic containment ('is a subset of').

The `subset` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `subset` element is an *n-ary set relation* (see Section 4.2.4).

Example

```
<apply>
  <subset/>
  <ci> A </ci>
  <ci> B </ci>
</apply>
```

Default Rendering

$A \subseteq B$

4.4.6.8 Proper Subset (`prsubset`)

Discussion

The `prsubset` element is the relational operator element for set-theoretic proper containment ('is a proper subset of').

The `prsubset` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `subset` element is an *n-ary set relation* (see Section 4.2.4).

Example

```
<apply>
  <prsubset/>
  <ci> A </ci>
  <ci> B </ci>
</apply>
```

Default Rendering

$A \subset B$

4.4.6.9 Not Subset (notsubset)

Discussion

The notsubset element is the relational operator element for the set-theoretic relation 'is not a subset of'.

The notsubset element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The notsubset element is a *binary set relation* (see Section 4.2.4).

Example

```
<apply>
  <notsubset/>
  <ci> A </ci>
  <ci> B </ci>
</apply>
```

Default Rendering

$$A \not\subseteq B$$

4.4.6.10 Not Proper Subset (notprsubset)

Discussion

The notprsubset element is the operator element for the set-theoretic relation 'is not a proper subset of'.

The notprsubset takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The notprsubset element is a *binary set relation* (see Section 4.2.4).

Example

```
<apply>
  <notprsubset/>
  <ci> A </ci>
  <ci> B </ci>
</apply>
```

Default Rendering

$$A \not\subset B$$

4.4.6.11 Set Difference (setdiff)

Discussion

The setdiff element is the operator element for a set-theoretic difference of two sets.

The setdiff element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The setdiff element is a *binary set operator* (see Section 4.2.3).

Example

```
<apply>
  <setdiff/>
  <ci> A </ci>
  <ci> B </ci>
</apply>
```

Default Rendering

$$A \setminus B$$

4.4.6.12 Cardinality (card)

Discussion

The card element is the operator element for deriving the size or cardinality of a set

The card element takes the attributes definitionURL, encoding that can be used to override the default semantics.

The card element is a *unary set operator* (see Section 4.2.3).

Example

```
<apply>
  <eq/>
  <apply><card/>
    <ci> A </ci>
  </apply>
  <ci> 5 </ci>
</apply>
```

where A is a set with 5 elements.

Default Rendering

$$|A|$$

4.4.7 Sequences and Series

4.4.7.1 Sum (sum)

Discussion

The sum element denotes the summation operator. Upper and lower limits for the sum, and more generally a domains for the bound variables are specified using uplimit, lowlimit or a condition on the bound variables. The index for the summation is specified by a bvar element.

The sum element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The sum element is an *operator taking qualifiers* (see Section 4.2.3.2).

Examples

```
<apply>
  <sum/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <lowlimit>
    <ci> a </ci>
  </lowlimit>
  <uplimit>
    <ci> b </ci>
  </uplimit>
  <apply><fn><ci> f </ci></fn>
    <ci> x </ci>
  </apply>
</apply>

<apply>
  <sum/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <condition>
    <apply> <in/>
      <ci> x </ci>
      <ci type="set"> B </ci>
    </apply>
  </condition>
  <apply><fn><ci> f </ci></fn>
    <ci> x </ci>
  </apply>
</apply>
```

Default Rendering

$$\sum_{x=a}^{b} f(x)$$

$$\sum_{x \in B} f(x)$$

4.4.7.2 Product (`product`)

Discussion

The `product` element denotes the product operator. Upper and lower limits for the product, and more generally a domains for the bound variables are specified using `uplimit`, `lowlimit` or a `condition` on the bound variables. The index for the product is specified by a `bvar` element.

The `product` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `product` element is an *operator taking qualifiers* (see Section 4.2.3.2).

Examples

```
<apply>
  <product/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <lowlimit>
    <ci> a </ci>
  </lowlimit>
  <uplimit>
    <ci> b </ci>
  </uplimit>
  <apply><fn><ci> f </ci></fn>
    <ci> x </ci>
  </apply>
</apply>

<apply>
  <product/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <condition>
    <apply> <in/>
      <ci> x </ci>
      <ci type="set"> B </ci>
    </apply>
  </condition>
  <apply><fn><ci> f </ci></fn>
    <ci> x </ci>
  </apply>
</apply>
```

Default Rendering

$$\prod_{x=a}^{b} f(x)$$

$$\prod_{x\in B} f(x)$$

4.4.7.3 Limit (`limit`)

Discussion

The `limit` element represents the operation of taking a limit of a sequence. The limit point is expressed by specifying a `lowlimit` and a `bvar`, or by specifying a `condition` on one or more bound variables.

The `limit` element takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

The `limit` element is an *operator taking qualifiers* (see Section 4.2.3.2).

Examples

```
<apply>
  <limit/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <lowlimit>
    <cn> 0 </cn>
  </lowlimit>
  <apply><sin/>
    <ci> x </ci>
  </apply>
</apply>
```

```
<apply>
  <limit/>
  <bvar>
    <ci> x </ci>
  </bvar>
  <condition>
    <apply>
      <tendsto type="above"/>
      <ci> x </ci>
      <ci> a </ci>
    </apply>
  </condition>
  <apply><sin/>
     <ci> x </ci>
  </apply>
</apply>
```

Default Rendering

- $\lim_{x \to 0} \sin x$
- $\lim_{x \downarrow a} \sin x$

4.4.7.4 Tends To (`tendsto`)

Discussion

The `tendsto` element is used to express the relation that a quantity is tending to a specified value.

The `tendsto` element takes the attributes `type` to set the direction from which the the limiting value is approached and the `definitionURL` and `encoding` attributes, which can be used to override the default semantics.

The `tendsto` element is a *binary relational operator* (see Section 4.2.4).

Examples

```
<apply>
  <tendsto type="above"/>
  <apply>
    <power/>
    <ci> x </ci>
    <cn> 2 </cn>
  </apply>
  <apply>
    <power/>
    <ci> a </ci>
    <cn> 2 </cn>
  </apply>
</apply>
```

To express $(x, y) \rightarrow (f(x, y), g(x, y))$, one might use vectors, as in:

```
<apply>
  <tendsto/>
  <vector>
     <ci> x </ci>
     <ci> y </ci>
  </vector>
  <vector>
    <apply><fn><ci> f </ci></fn>
      <ci> x </ci>
      <ci> y </ci>
    </apply>
    <apply><fn><ci> g </ci></fn>
      <ci> x </ci>
      <ci> y </ci>
    </apply>
  </vector>
</apply>
```

Default Rendering

$x^2 \downarrow a^2$ $(x,y) \rightarrow (f(x,y), g(x,y))$

4.4.8 Elementary classical functions

The names of the common trigonometric functions supported by MathML are listed below. Since their standard interpretations are widely known, they are discussed as a group.

`sin`	`cos`	`tan`
`sec`	`csc`	`cot`
`sinh`	`cosh`	`tanh`
`sech`	`csch`	`coth`
`arcsin`	`arccos`	`arctan`
`arccosh`	`arccot`	`arccoth`
`arccsc`	`arccsch`	`arcsec`
`arcsech`	`arcsinh`	`arctanh`

4.4.8.1 Discussion

These operator elements denote the standard trigonometrical functions.

These elements all take the `definitionURL` and `encoding` attributes, which can be used to override the default semantics.

They are all *unary trigonometric operators*. (see Section 4.2.3).

4.4.8.2 Examples

```
<apply>
  <sin/>
  <ci> x </ci>
</apply>
```

```
<apply>
  <sin/>
  <apply>
    <plus/>
    <apply><cos/>
      <ci> x </ci>
    </apply>
    <apply>
      <power/>
      <ci> x </ci>
      <cn> 3 </cn>
    </apply>
  </apply>
</apply>
```

4.4.8.3 Default Rendering

- $\sin x$
- $\sin(\cos x + x^3)$

4.4.8.4 Exponential (`exp`)

Discussion

The `exp` element represents the exponential function associated with the inverse of the `ln` function. In particular, exp(1) is approximately 2.718281828.

The `exp` element takes the `definitionURL` and `encoding`attributes, which may be used to override the default semantics.

The `exp` element is a *unary arithmetic operator* (see Section 4.2.3).

Example

```
<apply>
  <exp/>
  <ci> x </ci>
</apply>
```

Default Rendering

$$e^x$$

4.4.8.5 Natural Logarithm (ln)

Discussion

The ln element is the natural logarithm operator.

The ln element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The ln element is an *unary calculus operator* (see Section 4.2.3).

Example

```
<apply>
  <ln/>
  <ci> a </ci>
</apply>
```

If $a = e$ this will yield the value 1.

Default Rendering

$$\ln x$$

4.4.8.6 Logarithm (log)

Discussion

The log element is the operator that returns a logarithm to a given base. The base may be specified using a logbase element, which should be the first element following log, i.e. the second child of the containing apply element. If the logbase element is not present, a default base of 10 is assumed.

The log element takes the definitionURL and encodingattributes, which can be used to override the default semantics.

The log element can be used as either an *operator taking qualifiers* or a *unary calculus operator* (see Section 4.2.3.2).

Example

```
<apply>
  <log/>
  <logbase>
    <cn> 3 </cn>
  </logbase>
  <ci> x </ci>
</apply>
```

This markup represents ‘the base 3 logarithm of x’. For natural logarithms base e, the `ln` element should be used instead.

Default Rendering

$$\log_3 x$$

4.4.9 Statistics

4.4.9.1 Mean (`mean`)

Discussion

`mean` is the operator element for a mean or average.

`mean` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

Example

`mean` is an n-ary operator (see Section 4.2.3).

```
<apply>
  <mean/>
  <ci> X </ci>
</apply>
```

Default Rendering

$\bar{X}$or $\langle X \rangle$

4.4.9.2 Standard Deviation (`sdev`)

Discussion

`sdev` is the operator element for the standard deviation.

`sdev` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

Example

`sdev` is an n-ary operator (see Section 4.2.3).

```
<apply>
  <sdev/>
  <ci> X </ci>
</apply>
```

Default Rendering

$$\sigma(X)$$

4.4.9.3 Variance (variance)

Discussion

variance is the operator element for the statistical variance.

variance takes the definitionURL and encodingattributes, which can be used to override the default semantics.

Example

variance is an n-ary operator (see Section 4.2.3).

```
<apply>
  <variance/>
  <ci> X </ci>
</apply>
```

Default Rendering

$$\sigma(X)^2$$

4.4.9.4 Median (median)

Discussion

median is the operator element for the median.

median takes the definitionURL and encodingattributes, which can be used to override the default semantics.

Example

median is an n-ary operator (see Section 4.2.3).

```
<apply>
  <median/>
  <ci> X </ci>
</apply>
```

Default Rendering

$$\mathrm{median}(X)$$

4.4.9.5 Mode (mode)

Discussion

mode is the operator for the statistical mode.

mode takes the definitionURL and encodingattributes, which can be used to override the default semantics.

Example

`mode` is an n-ary operator (see Section 4.2.3).

```
<apply>
  <mode/>
  <ci> X </ci>
</apply>
```

Default Rendering

$$\text{mode}(X)$$

4.4.9.6 Moment (`moment`)

Discussion

The `moment` element represents statistical moments. Use `degree` for the n in ‘ n-th moment’.

`moment` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

Example

`moment` is an operator taking qualifiers (see Section 4.2.3.2).

```
<apply>
  <moment/>
  <degree>
    <cn> 3 </cn>
  </degree>
  <ci> X </ci>
</apply>
```

Default Rendering

$$\langle X^3 \rangle$$

4.4.10 Linear Algebra

4.4.10.1 Vector (`vector`)

Discussion

`vector` is the container element for a vector. The child elements form the components of the vector.

For purposes of interaction with matrices and matrix multiplication, vectors are regarded as equivalent to a matrix consisting of a single column, and the transpose of a vector behaves the same as a matrix consisting of a single row.

Example

`vector` is a constructor element (see Section 4.2.2.2).

```
<vector>
  <cn> 1 </cn>
  <cn> 2 </cn>
  <cn> 3 </cn>
  <ci> x </ci>
</vector>
```

Default Rendering

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ x \end{pmatrix}$$

$(1, 2, 3, x)$

4.4.10.2 Matrix (`matrix`)

Discussion

The `matrix` element is the container element for matrix rows, which are represented by `matrixrow`. The `matrixrows` contain the elements of a matrix.

Example

`matrix` is a constructor element (see Section 4.2.2.2).

```
<matrix>
  <matrixrow>
    <cn> 0 </cn> <cn> 1 </cn> <cn> 0 </cn>
  </matrixrow>
  <matrixrow>
    <cn> 0 </cn> <cn> 0 </cn> <cn> 1 </cn>
  </matrixrow>
  <matrixrow>
    <cn> 1 </cn> <cn> 0 </cn> <cn> 0 </cn>
  </matrixrow>
</matrix>
```

Default Rendering

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

4.4.10.3 Matrix row (`matrixrow`)

Discussion

The `matrixrow` element is the container element for the rows of a matrix.

Example

`matrixrow` is a constructor element (see Section 4.2.2.2).

```
<matrixrow>
  <cn> 1 </cn>
  <cn> 2 </cn>
</matrixrow>
<matrixrow>
  <cn> 3 </cn>
  <ci> x </ci>
</matrixrow>
```

Default Rendering

Matrix rows are not directly rendered by themselves outside of the context of a matrix.

4.4.10.4 Determinant (`determinant`)

Discussion

The `determinant` element is the operator for constructing the determinant of a matrix.

`determinant` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

Example

`determinant` is a *unary operator* (see Section 4.2.3).

```
<apply>
  <determinant/>
  <ci type="matrix"> A </ci>
</apply>
```

Default Rendering

$$\det A$$

4.4.10.5 Transpose (`transpose`)

Discussion

The `transpose` element is the operator for constructing the transpose of a matrix.

`transpose` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

Example

`transpose` is a *unary operator* (see Section 4.2.3).

```
<apply>
  <transpose/>
  <ci type="matrix"> A </ci>
</apply>
```

Default Rendering

A^{t}

4.4.10.6 Selector (`selector`)

Discussion

The `selector` element is the operator for indexing into vectors matrices and lists. It accepts one or more arguments. The first argument identifies the vector, matrix or list from which the selection is taking place, and the second and subsequent arguments, if any, indicate the kind of selection taking place.

When `selector` is used with a single argument, it should be interpreted as giving the sequence of all elements in the list, vector or matrix given. The ordering of elements in the sequence for a matrix is understood to be first by column, then by row. That is, for a matrix ($a_{i,j}$), where the indices denote row and column, the ordering would be $a_{1,1}$, $a_{1,2}$, ... $a_{2,1}$, $a_{2,2}$ ... etcetera.

When three arguments are given, the last one is ignored for a list or vector, and in the case of a matrix, the second and third arguments specify the row and column of the selected element.

When two arguments are given, and the first is a vector or list, the second argument specifies an element in the list or vector. When a matrix and only one index i is specified as in

```
<apply>
  <selector/>
  <matrix>
    <matrixrow>
      <cn> 1 </cn> <cn> 2 </cn>
    </matrixrow>
    <matrixrow>
      <cn> 3 </cn> <cn> 4 </cn>
    </matrixrow>
  </matrix>
  <cn> 1 </cn>
</apply>
```

it refers to the i-th matrixrow. Thus, the preceding example selects the following row:

```
<matrixrow> <cn> 1 </cn> <cn> 2 </cn> </matrixrow>
```

`selector` takes the `definitionURL` and `encoding`attributes, which can be used to override the default semantics.

`selector` is classified as an n-ary linear algebra operator even though it can take only one, two, or three arguments.

Example

```
<apply>
  <selector/>
  <ci type="matrix"> A </ci>
  <cn> 3 </cn>
  <cn> 2 </cn>
</apply>
```

Default Rendering

The `selector` construct renders the same as the expression it selects.

4.4.10.7 Vector product (vectorproduct)

Discussion

The vectorproduct is the operator element for deriving the vector product of two vectors

The vectorproduct element takes the attributes definitionURL, encoding that can be used to override the default semantics.

The vectorproduct element is a *binary vector operator* (see Section 4.2.3).

Example

```
<apply>
  <eq/>
  <apply><vectorproduct/>
    <ci type="vector"> A </ci>
    <ci type="vector"> B </ci>
  </apply>
  <apply><times/>
    <ci> a </ci>
    <ci> b </ci>
    <apply><sin/>
      <ci> &theta; </ci>
    </apply>
  </apply>
</apply>
```

where A and B are vectors, a, b are the magnitudes of A, B and θis the angle between A and B.

Default Rendering

$$A \times B$$

4.4.10.8 Scalar product (scalarproduct)

Discussion

The scalarproduct is the operator element for deriving the scalar product of two vectors

The scalarproduct element takes the attributes definitionURL, encoding that can be used to override the default semantics.

The scalarproduct element is a *binary vector operator* (see Section 4.2.3).

Example

```
<apply>
  <eq/>
  <apply><scalarproduct/>
    <ci type="vector"> A </ci>
    <ci type="vector">B </ci>
  </apply>
  <apply><times/>
    <ci> a </ci>
    <ci> b </ci>
    <apply><cos/>
      <ci> &theta; </ci>
    </apply>
  </apply>
</apply>
```

where A and B are vectors, a, b are the magnitudes of A, B and θis the angle between A and B.

Default Rendering

$A.B$

4.4.10.9 Outer product (`outerproduct`)

Discussion

The `outerproduct` is the operator element for deriving the outer product of two vectors

The `outerproduct` element takes the attributes `definitionURL, encoding` that can be used to override the default semantics.

The `outerproduct` element is a *binary vector operator* (see Section 4.2.3).

Example

```
<apply>
  <outerproduct/>
  <ci type="vector">A</ci>
  <ci type="vector">B</ci>
</apply>
```

where A and B are vectors.

Default Rendering

$A.B$

4.4.11 Semantic Mapping Elements

This section explains the use of the semantic mapping elements `semantics`, `annotation` and `annotation-xml`.

4.4.11.1 Annotation (annotation)

Discussion

The annotation element is the container element for a semantic annotation in a non-XML format.

The annotation element takes the attribute encoding to define the encoding being used.

Example

The annotation element is a semantic mapping element. It is always used with semantics.

```
<semantics>
  <apply>
    <plus/>
    <apply><sin/>
      <ci> x </ci>
    </apply>
    <cn> 5 </cn>
  </apply>
  <annotation encoding="TeX">
    \sin x + 5
  </annotation>
</semantics>
```

Default Rendering

None. The information contained in annotations may optionally be used by a renderer able to process the kind of annotation given.

4.4.11.2 Semantics (semantics)

Discussion

The semantics element is the container element that associates additional representations with a given MathML construct. The semantics element has as its first child the expression being annotated, and the subsequent children are the annotations. There is no restriction on the kind of annotation that can be attached using the semantics element. For example, one might give a TEX encoding, or computer algebra input in an annotation.

The representations that are XML based are enclosed in an annotation-xml element while those representations that are to be parsed as PCDATA are enclosed in an annotation element.

The semantics element takes the definitionURL and encodingattributes, which can be used to reference an external source for some or all of the semantic information.

An important purpose of the semantics construct is to associate specific semantics with a particular presentation, or additional presentation information with a content construct. The default rendering of a semantics element is the default rendering of its first child. When a MathML-presentation annotation is provided, a MathML renderer may optionally use this information to render the MathML construct. This would typically be the case when the first child is a MathML content construct and the annotation is provided to give a preferred rendering differing from the default for the content elements.

Use of semantics to attach additional information in-line to a MathML construct can be contrasted with use of the csymbol for referencing external semantics. See Section 4.4.1.3

Examples

The `semantics` element is a semantic mapping element.

```
<semantics>
  <apply>
    <plus/>
    <apply>
      <sin/>
      <ci> x </ci>
    </apply>
    <cn> 5 </cn>
  </apply>
  <annotation encoding="Maple">
    sin(x) + 5
  </annotation>
  <annotation-xml encoding="MathML-Presentation">
    ...
    ...
  </annotation-xml>
  <annotation encoding="Mathematica">
    Sin[x] + 5
  </annotation>
  <annotation encoding="TeX">
    \sin x + 5
  </annotation>
  <annotation-xml encoding="OpenMath">
    <OMA>...</OMA>
  </annotation-xml>
</semantics>
```

Default Rendering

The default rendering of a `semantics` element is the default rendering of its first child.

4.4.11.3 XML-based annotation (`annotation-xml`)

Discussion

The `annotation-xml` container element is used to contain representations that are XML based. It is always used together with the `semantics` element, and takes the attribute `encoding` to define the encoding being used.

`annotation-xml` is a semantic mapping element.

Example

```
<semantics>
  <apply>
    <plus/>
    <apply><sin/>
      <ci> x </ci>
    </apply>
    <cn> 5 </cn>
  </apply>
  <annotation-xml encoding="OpenMath">
    <OMA><OMS name="plus" cd="arith1"/>
      <OMA><OMS name="sin" cd="transc1"/>
        <OMV name="x"/>
      </OMA>
      <OMI>5</OMI>
    </OMA>
  </annotation-xml>
</semantics>
```

See also the discussion of `semantics` above.

Default Rendering

None. The information may optionally be used by a renderer able to process the kind of annotation given.

4.4.12 Constant and Symbol Elements

This section explains the use of the Constant and Symbol elements.

4.4.12.1 integers (`integers`)

Discussion

`integers` represents the set of all integers.

Example

```
<apply>
  <in/>
  <cn type="integer"> 42 </cn>
  <integers/>
</apply>
```

Default Rendering

$$42 \in \mathbb{Z}$$

4.4.12.2 reals (`reals`)

Discussion

`reals` represents the set of all real numbers.

Example

```
<apply>
  <in/>
  <cn type="real"> 44.997 </cn>
  <reals/>
</apply>
```

Default Rendering

$44.997 \in \mathbb{R}$

4.4.12.3 rationals (reals)

Discussion

rationals represents the set of all rational numbers.

Example

```
<apply>
  <in/>
  <cn type="rational"> 22 <sep/>7</cn>
  <rationals/>
</apply>
```

Default Rendering

$22/7 \in \mathbb{Q}$

4.4.12.4 naturalnumbers (naturalnumbers)

Discussion

naturalnumbers represents the set of all natural numbers, ie. non-negative integers.

Example

```
<apply>
  <in/>
  <cn type="integer">1729</cn>
  <naturalnumbers/>
</apply>
```

Default Rendering

$1729 \in \mathbb{N}$

4.4.12.5 complexes (`complexes`)

Discussion

`complexes` represents the set of all complex numbers, ie. numbers which may have a real and an imaginary part.

Example

`complexes` represents the set of all complex numbers, ie. numbers which may have a real and an imaginary part.

Example

```
<apply>
  <in/>
  <ci type="complex">17<sep/>29</ci>
  <complexes/>
</apply>
```

Default Rendering

$$17 + 29i \in \mathbb{C}$$

4.4.12.6 primes (`primes`)

Discussion

`primes` represents the set of all natural prime numbers, ie. integers greater than 1 which have no positive integer factor other than themselves and 1.

Example

```
<apply>
  <in/>
  <cn type="integer">17</cn>
  <primes/>
</apply>
```

Default Rendering

$$17 \in \mathbb{P}$$

4.4.12.7 exponentiale (`exponentiale`)

Discussion

`exponentiale` represents the mathematical constant which is the exponential base of the natural logarithms, commonly written *e . It is approximately 2.718281828..*

Example

```
<apply> <eq/>
  <apply>
    <ln/>
    <exponentiale/>
  </apply>
  <cn>1</cn>
</apply>
```

Default Rendering

$$ln e = 1$$

4.4.12.8 imaginaryi (`imaginaryi`)

Discussion

`imaginaryi` represents the mathematical constant which is the square root of -1, commonly written *i*.

Example

```
<apply> <eq/>
  <apply>
    <power/>
    <imaginaryi/>
    <cn>2</cn>
  </apply>
  <cn>-1</cn>
</apply>
```

Default Rendering

$$i^2 = -1$$

4.4.12.9 notanumber (`notanumber`)

Discussion

`notanumber` represents the result of an ill-defined floating point operation, sometimes also called *NaN*.

Example

```
<apply> <eq/>
  <apply>
    <divide/>
    <cn>0</cn>
    <cn>0</cn>
  </apply>
  <notanumber/>
</apply>
```

Default Rendering

$$0/0 = NaN$$

4.4.12.10 true (`true`)

Discussion

`true` represents the logical constant for truth.

Example

```
<apply> <eq/>
  <apply>
    <or/>
    <true/>
    <ci type = "logical">P</ci>
  </apply>
  <true/>
</apply>
```

Default Rendering

$$true or P = true$$

4.4.12.11 false (`false`)

Discussion

`false` represents the logical constant for falsehood.

Example

```
<apply> <eq/>
  <apply>
    <and/>
    <false/>
    <ci type = "logical">P</ci>
  </apply>
  <false/>
</apply>
```

Default Rendering

$$false and P = false$$

4.4.12.12 emptyset (`emptyset`)

Discussion

`emptyset` represents the empty set.

Example

```
<apply>
  <neq/>
  <integers/>
  <emptyset/>
</apply>
```

Default Rendering

$$\mathbb{Z} \neq \emptyset$$

4.4.12.13 pi (`pi`)

Discussion

`pi` represents the mathematical constant which is the ratio of a circle's circumference to its diameter, approximately 3.141592653.

Example

```
<apply>
  <approx/>
  <pi/>
  <cn type = "rational">22<sep/>7</cn>
</apply>
```

Default Rendering

$$\pi \approx 22/7$$

4.4.12.14 eulergamma (`eulergamma`)

Discussion

`eulergamma` represents Euler's constant, approximately 0.5772156649

Example

```
<eulergamma/>
```

Default Rendering

$$\Gamma$$

4.4.12.15 infinity (`infinity`)

Discussion

`infinity` represents the concept of infinity. Proper interpretation depends on context.

Example

```
<infinity/>
```

Default Rendering

$$\infty$$

Chapter 5

Combining Presentation and Content Markup

Presentation markup and content markup can be combined in two ways. The first manner is to intersperse content and presentation elements in what is essentially a single tree. This is called *mixed* markup. The second manner is to provide *both* an explicit presentation and an explicit content in a pair of trees. This is called *parallel* markup. This chapter describes both mixed and parallel markup, and how they may used in conjunction with style sheets and other tools.

5.1 Why Two Different Kinds of Markup?

Chapters 3 and 4 describe two kinds of markup for encoding mathematical material in documents.

Presentation markup captures *notational structure*. It encodes the notational structure of an expression in a sufficiently abstract way to facilitate rendering to various media. Thus, the same presentation markup can be rendered with relative ease on screen in either wide and narrow windows, in ASCII or graphics, in print, or it can be enunciated in a sensible way when spoken. It does this by providing information such as structured grouping of expression parts, classification of symbols, etc.

Presentation markup does *not* directly concern itself with the mathematical structure or meaning of an expression. In many situations, notational structure and mathematical structure are closely related, so a sophisticated processing application may be able to heuristically infer mathematical meaning from notational structure, provided sufficient context is known. However, in practice, the inference of mathematical meaning from mathematical notation must often be left to the reader.

Employing presentation tags alone may limit the ability to re-use a MathML object in another context, especially evaluation by external applications.

Content markup captures *mathematical structure*. It encodes mathematical structure in a sufficiently regular way in order to facilitate the assignment of mathematical meaning to an expression by application programs. Though the details of mapping from mathematical expression structure to mathematical meaning can be extremely complex, in practice, there is wide agreement about the conventional meaning of many basic mathematical constructions. Consequently, much of the meaning of a content expression is easily accessible to a processing application, independently of where or how it is displayed to the reader. In many cases, content markup could be cut from a Web browser and pasted into a mathematical software tool with confidence that sensible values will be computed.

Since content markup is *not* directly concerned with how an expression is displayed, a renderer must infer how an expression should be presented to a reader. While a sufficiently sophisticated renderer and style-sheet mechanism could in principle allow a user to read mathematical documents using personalized notational preferences, in practice, rendering content expressions with notational nuances may still require intervention of some sort.

Employing content tags alone may limit the ability of the author to precisely control how an expression is rendered.

Both content and presentation tags are necessary in order to provide the full expressive capability one would expect in a mathematical markup language. Often the same mathematical notation is used to represent several completely different concepts. For example, the notation x^i may be intended (in polynomial algebra) as the i-th power of the variable x, or as the i-th component of a vector x (in tensor calculus). In other cases, the same mathematical concept may be displayed in one of various notations. For instance, the factorial of a number might be expressed with an exclamation mark, a Gamma function, or a Pochhammer symbol.

Thus the same notation may represent several mathematical ideas, and, conversely, the same mathematical idea often has several notations. In order to provide authors with the ability to precisely control notation while at the same time encoding meanings in a machine-readable way, both content and presentation markup are needed.

In general, if it is important to control exactly how an expression is rendered, presentation markup will generally be more satisfactory. If it is important that the meaning of an expression can be interpreted dependably and automatically, then content markup will generally be more satisfactory.

5.2 Mixed Markup

MathML offers authors elements for both content and presentation markup. Whether to use one or the other, or a combination of both, depends on what aspects of rendering and interpretation an author wishes to control, and what kinds of re-use he or she wishes to facilitate.

5.2.1 Reasons to Mix Markup

In many common situations, an author or authoring tool may choose to generate either presentation or content markup exclusively. For example, a program for translating legacy documents would most likely generate pure presentation markup. Similarly, an educational software package might very well generate only content markup for evaluation in a computer algebra system. However, in many other situations, there are advantages to mixing both presentation and content markup within a single expression.

If an author is primarily presentation-oriented, interspersing some content markup will often produce more accessible, more re-usable results. For example, an author writing about linear algebra might write:

```
<mrow>
  <apply>
    <power/>
    <ci>x</ci><cn>2</cn>
  </apply>
  <mo>+</mo>
  <msup>
    <mi>v</mi><mn>2</mn>
  </msup>
</mrow>
```

where v is a vector and the superscript denotes a vector component, and x is a real variable. On account of the linear algebra context, a visually impaired reader may have directed his or her voice synthesis software to render superscripts as vector components. By explicitly encoding the power, the content markup yields a much better voice rendering than would likely happen by default.

If an author is primarily content-oriented, there are two reasons to intersperse presentation markup. First, using presentation markup provides a way of modifying or refining how a content expression is rendered. For example, one might write:

```
<apply>
  <in/>
  <ci><mi fontweight="bold">v</mi></ci>
  <ci>S</ci>
</apply>
```

In this case, the use of embedded presentation markup allows the author to specify that v should be rendered in boldface. In the same way, it is somtimes the case that a completely different notation is desired for a content expression. For example, here we express a fact about factorials, $n = n!/(n\text{-}1)!$, using the ascending factorial notation:

```
<apply>
  <equivalent/>
  <ci>n</ci>
  <apply>
     <divide/>
     <semantics>
        <apply>
           <factorial/>
           <ci>n</ci>
        </apply>
        <annotation-xml encoding="MathML-Presentation">
           <msup>
              <mn>1</mn>
              <mover accent="true">
                 <mi>n</mi>
                 <mo><mchar name="OverBar"/></mo>
              </mover>
           </msup>
        </annotation-xml>
     </semantics>
     <semantics>
        <apply>
           <factorial/>
           <apply><minus/><ci>n</ci><cn>1</cn></apply>
        </apply>
        <annotation-xml encoding="MathML-Presentation">
           <msup>
              <mn>1</mn>
              <mover accent="true">
                 <mrow><mi>n</mi><mo>-</mo><mn>1</mn></mrow>
                 <mo><mchar name="OverBar"/></mo>
              </mover>
           </msup>
        </annotation-xml>
     </semantics>
   </apply>
</apply>
```

This content expresison would render using the given notation as: $\frac{1^{\bar{n}}}{1^{\overline{n-1}}}$

A second reason to use presentation within content markup is that there is a continually growing list of areas of discourse that do not have pre-defined content elements for encoding their objects and operators. As a consequence, any system of content markup inevitably requires an extension mechanism that combines notation with semantics in some way. MathML content markup specifies several ways of attaching an external semantic definitions to content objects. It is necessary, however, to use MathML presentation markup to specify how such user-defined semantic extensions should be rendered.

For example, the ‘rank’ operator from linear algebra is not included as a pre-defined MathML content element. Thus, to express the statement rank($u^T v$)=1 we use a `semantics` element to bind a semantic definition to the symbol *rank*.

```
<apply>
  <eq/>
  <apply>
    <fn>
      <semantics>
        <mi>rank</mi>
        <annotation-xml encoding="OpenMath">
          <OMS cd="linalg3" name="rank"/>
        </annotation-xml>
      </semantics>
    </fn>
    <apply>
      <times/>
      <apply> <transpose/> <ci>u</ci> </apply>
      <ci>v</ci>
    </apply>
  </apply>
  <cn>1</cn>
</apply>
```

Here, the semantics of rank have been given using a symbol from an OpenMath content dictionary (CD).

5.2.2 Combinations that are prohibited

The main consideration when presentation markup and content markup are mixed together in a single expression is that the result should still make sense. When both kinds of markup are contained in a presentation expression, this means it should be possible to render the resulting mixed expressions simply and sensibly. Conversely, when mixed markup appears in a content expression, it should be possible to simply and sensibly assign a semantic interpretation to the expression as whole. These requirements place a few natural constraints on how presentation and content markup can be mixed in a single expression, in order to avoid ambiguous or otherwise problematic expressions.

Two examples illustrate the kinds of problems that must be avoided in mixed markup. Consider:

```
<mrow>
  <bvar> x </bvar> <mo> + </mo> <bvar> y </bvar>
</mrow>
```

In this example, the content element `bvar` has been indiscriminately embedded in a presentation expression. Since `bvar` requires an enclosing context for its meaning, this expression is unclear.

Similarly, consider:

```
<apply>
  <ci> x </ci> <mo> + </mo> <ci> y </ci>
</apply>
```

Here, the `mo` element is problematic. Should a renderer infer that the usual arithmetic operator is intended, and act as if the prefix content element `plus` had been used? Or should this be literally interpreted as the operator *x* applied to two arguments, `<mo>+</mo>` and `<mi>y</mi>` ? Even if we were to decide that `<mo>+</mo>` was the operator, then what should its meaning be? These questions do not have particularly compelling answers, so this kind of mixing of content and presentation markup is also prohibited.

5.2.3 Presentation Markup Contained in Content Markup

The use of presentation markup within content markup is limited to situations that do not effect the ability of content markup to unambiguously encode mathematical meaning. Specifically, presentation markup may only appear in content markup in three ways:

1. within `ci` and `cn` token elements
2. within the `csymbol` element
3. within the `semantics` element

Any other presentation markup occurring within a content markup is a MathML error. More detailed discussion of these three cases follows:

Presentation markup within token elements. The token elements `ci` and `cn` are permitted to contain any sequence of MathML characters (defined in Chapter 6), presentation elements, and `sep` empty elements. Contiguous blocks of MathML characters in `ci` and `cn` elements are rendered as if they were wrapped in `mi` and `mn` elements respectively. If a token element contains both MathML characters and presentation elements, contiguous blocks of MathML characters (if any) are treated as if wrapped in `mi` or `mn` elements as appropriate, and the resulting collection of presentation elements are rendered as if wrapped in an `mrow` element.

Presentation markup within the `csymbol` element. The `csymbol` element may contain either MathML characters interspersed with presentation markup, or content elements of the container type. It is a MathML error for a `csymbol` element to contain both presentation and content elements. When the `csymbol` element contains both raw data and presentation markup, the same rendering rules that apply to content elements of the token type should be used.

Presentation markup within the `semantics` element. One of the main purposes of the `semantics` element is to provide a mechanism for incorporating arbitrary MathML expressions into content markup in a semantically meaningful way. In particular, any valid presentation expression can be embedded in a content expression by placing it as the first child of a `semantics` element. The meaning of this wrapped expression should be indicated by one or more annotation elements also contained in the `semantics` element. Suggested rendering for a `semantics` element is discussed in Section 4.2.6.

5.2.4 Content Markup Contained in Presentation Markup

The guiding principle for embedding content markup within presentation expressions is that the resulting expression should still have an unambiguous rendering. In general, this means that embedded content expressions must be semantically meaningful, since rendering of content markup depends on its meaning.

Certain content elements derive part of their semantic meaning from the surrounding context, such as whether a `bvar` element is qualifying an integral, logical quantifier or lambda expression. Another example would be whether a `degree` element occurs in a `root` or `partialdiff` element. Thus, in a presentation context, elements such as these do not have a clearly defined meaning, and hence there is no obvious choice for a rendering. Consequently, they are not allowed.

Using the terminology of Section 4.2.1, we see that operator, relation, container, constant and symbol elements make sense on their own, while elements of the qualifier and condition type do not. (Note that `interval` may be used either as a general container, or as a qualifier.)

Outside these categories, certain elements deserve specific comment: the elements `declare`, `sep`, `annotation` and `annotation-xml` can only appear in very specific contexts and consequently are not permitted as direct sub-expressions of any presentation element. Finally, the element `semantics` carries with it sufficient information to be permitted in presentation.

The complete list of content elements that *cannot* appear as a child in a presentation element is: `annotation`, `annotation-xml`, `sep`, `declare`, `bvar`, `condition`, `degree`, `logbase`, `lowlimit`, `uplimit`.

Note that within presentation markup, content expressions may only appear in locations where it is valid for any MathML expression to appear. In particular, content expressions may not appear within presentation token elements. In this regard mixing presentation and content are asymmetrical.

Note that embedding content markup in presentation will often require applications to render operators outside of an `apply` context. E.g., it may be necessary to render `abs`, `plus`, `root` or `sin` outside of an application. Content/presentation mixing does not introduce any new requirements, however, since unapplied operators are already permitted in content expressions, for example:

```
<apply>
   <compose/>
   <sin/>
   <apply>
      <inverse/>
      <root/>
   </apply>
</apply>
```

5.3 Parallel Markup

Some applications are able to make use of *both* presentation and content information. For these applications it is desirable to provide both forms of markup for the same mathematical expression. This is called *parallel markup*.

Parallel markup is achieved with the `semantics` element. Parallel markup for an expression can be used on its own, or can be incorporated as part of a larger content or presentation tree.

5.3.1 Top-level Parallel Markup

In many cases what is desired is to provide presentation markup and content markup for a mathematical expression as a whole. To achieve this, a single `semantics` element is used pairing two markup trees, with the first branch being the MathML presentation markup, and the second branch being the MathML content markup.

The following example encodes the boolean arithmetic expression $(a+b)(c+d)$ in this way.

```
<semantics>
  <mrow>
    <mrow><mo>(</mo><mi>a</mi> <mo>+</mo> <mi>b</mi><mo>)</mo></mrow>
    <mo>&InvisibleTimes;</mo>
    <mrow><mo>(</mo><mi>c</mi> <mo>+</mo> <mi>d</mi><mo>)</mo></mrow>
  </mrow>
  <annotation-xml encoding="MathML-Content">
    <apply><and/>
      <apply><xor/><ci>a</ci> <ci>b</ci></apply>
      <apply><xor/><ci>c</ci> <ci>d</ci></apply>
    </apply>
  </annotation-xml>
</semantics>
```

This example is non-trivial in the sense that the content markup could not be easily derived from the presentation markup alone.

5.3.2 Fine-grained Parallel Markup

Top-level pairing of independent presentation and content markup is sufficient for many, but not all, situations. Applications that allow treatment of *sub-expressions* of mathematical objects require the ability to associate presentation, content or information with the *parts* of an object with mathematical markup. Top-level pairing with a `semantics` element is insufficient in this type of situation; identification of a sub-expression in one branch of `semantics` element gives no indication of the corresponding parts in other branches.

The ability to identify corresponding sub-expressions is required in applications such as mathematical expression editors. In this situation, selecting a sub-expression on a visual display can identify a particular portion of a presentation markup tree. The application then needs to determine the corresponding annotations of the sub-expressions; in particular, the application requires the sub-expressions of the `annotation-xml` tree in MathML content notation.

It is, in principle, possible to provide annotations for each presentation node by incorporating `semantics` elements recursively.

```
<semantics>
  <mrow>
    <semantics>
      <mrow><mo>(</mo><mi>a</mi> <mo>+</mo> <mi>b</mi><mo>)</mo></mrow>
      <annotation-xml encoding="MathML-Content">
        <apply><plus/><ci>a</ci> <ci>b</ci></apply>
      </annotation-xml>
    </semantics>
    <mo>&InvisibleTimes;</mo>
    <semantics>
      <mrow><mo>(</mo><mi>c</mi> <mo>+</mo> <mi>d</mi><mo>)</mo></mrow>
      <annotation-xml encoding="MathML-Content">
        <apply><plus/><ci>c</ci> <ci>d</ci></apply>
      </annotation-xml>
    </semantics>
  </mrow>

  <annotation-xml encoding="MathML-Content">
    <apply><times/>
      <apply><plus/><ci>a</ci> <ci>b</ci></apply>
      <apply><plus/><ci>c</ci> <ci>d</ci></apply>
    </apply>
  </annotation-xml>
</semantics>
```

To be complete this example would be much more verbose, wrapping each of the individual leaves `mi`, `mo` and `mn` in a further seven `semantics` elements.

This approach is very general and works for all kinds of annotations (including non-MathML annotations and multiple annotations). It leads, however, to O(n^2) increase in size of the document. This is therefore not a suitable approach for fine-grained parallel markup of large objects.

5.3.3 Parallel Markup via Cross-References: `id` and `xref`

To better accomodate applications that must deal with sub-expressions of large objects, MathML uses cross-references between the branches of a `semantics` element to identify corresponding sub-structures.

Cross-referencing is achieved using `id` and `xref` attributes within the branches of a containing `semantics` element. These attributes may optionally be placed on MathML elements of any type.

The following example shows this cross-referencing for the boolean arithmetic expression $(a+b)(c+d)$.

```
<semantics>
  <mrow id="E">
    <mrow id="E.1">
      <mo id="E.1.1">(</mo>
      <mi id="E.1.2">a</mi>
      <mo id="E.1.3">+</mo>
      <mi id="E.1.4">b</mi>
      <mo id="E.1.5">)</mo>
    </mrow>
    <mo id="E.2">&InvisibleTimes;</mo>
    <mrow id="E.3">
      <mo id="E.3.1">(</mo>
      <mi id="E.3.2">c</mi>
      <mo id="E.3.3">+</mo>
      <mi id="E.3.4">d</mi>
      <mo id="E.3.5">)</mo>
    </mrow>
  </mrow>

  <annotation-xml encoding="MathML-Content">
    <apply xref="E">
      <and xref="E.2"/>
      <apply xref="E.1">
        <xor xref="E.1.3"/><ci xref="E.1.2">a</ci><ci xref="E.1.4">b</ci>
      </apply>
      <apply xref="E.3">
        <xor xref="E.3.3"/><ci xref="E.3.2">c</ci><ci xref="E.3.4">d</ci>
      </apply>
    </apply>
  </annotation-xml>
</semantics>
```

An `id` attribute and a corresponding `xref` appearing within the same `semantics` element create a correspondence between sub-expressions.

In creating these correspondences by cross-reference, *all* of the `id` attributes referenced by any `xref` must be in the *same* branch of an enclosing `semantics` element. This constraint guarantees that these correspondences do not create unintentional cycles. (Note that this restriction does *not* exclude the use of `id` attributes within the other branches of the enclosing `semantics` element. It does, however, exclude references to these other `id` attributes originating in the same `semantics` element.)

There is no restriction on which branch of the `semantics` element may contain the destination `id` attributes. It is up to the application to determine which branch to use.

In general, there will not be a one-to-one correspondence between nodes in parallel branches. For example, a presentation tree may contain elements, such as parentheses, that have no correspondents in the content tree. It is therefore often useful to put the `id` attributes on the branch with the finest-grained node structure. Then all of the other branches will have `xref` attributes to some subset of the `id` attributes.

In absence of other criteria, the first branch of the `semantics` element is a sensible choice to contain the `id` attributes. Applications that add or remove annotations will then not have to re-assign attributes to the `semantics` trees.

In general, the use of `id` and `xref` attributes allows a full correspondence between sub-expressions to be given in text that is at most a constant factor larger than the original. The direction of the references should not be taken to imply that sub-expression selection is intended to be permitted only on one child of the `semantics` element. It is equally feasible to select a subtree in any branch and to recover the corresponding subtrees of the other branches.

5.3.4 Annotation Cross-References using XLink: `id` and `href`

It is possible to give cross-references between a MathML expression and a non-MathML XML annotation using the XLink protocol [XLink]. As an example, the boolean expression of the previous section can be annotated with OpenMath, and cross-linked as follows:

```
<semantics>
  <mrow id="E">
    <mrow id="E.1" xlink:id="E.1">
      <mo id="E.1.1">(</mo>
      <mi id="E.1.2">a</mi>
      <mo id="E.1.3">+</mo>
      <mi id="E.1.4">b</mi>
      <mo id="E.1.5">)</mo>
    </mrow>
    <mo id="E.2">&InvisibleTimes;</mo>
    <mrow id="E.3">
      <mo id="E.3.1">(</mo>
      <mi id="E.3.2">c</mi>
      <mo id="E.3.3">+</mo>
      <mi id="E.3.4">d</mi>
      <mo id="E.3.5">)</mo>
    </mrow>
  </mrow>

  <annotation-xml encoding="MathML-Content">
    <apply xref="E">
      <and xref="E.2"/>
      <apply xref="E.1">
        <xor xref="E.1.3"/><ci xref="E.1.2">a</ci><ci xref="E.1.4">b</ci>
      </apply>
      <apply xref="E.3">
        <xor xref="E.3.3"/><ci xref="E.3.2">c</ci><ci xref="E.3.4">d</ci>
      </apply>
    </apply>
  </annotation-xml>

  <annotation-xml encoding="OpenMath">
    <OMA xlink:href="id('E')" xmlns="www.openmath.org/OpenMath">
      <OMS cd="logic1" name="and" xlink:href="id('E')"/>
      <OMA xlink:href="id('E.1')">
        <OMS cd="logic1" name="xor" xlink:href="id('E.1.3')"/>
        <OMV name="a" xlink:href="id('E.1.2')"/>
        <OMV name="b" xlink:href="id('E.1.4')"/>
      </OMA>
```

```
      <OMA xlink:href="id('E.3')">
        <OMS cd="logic1" name="xor" xlink:href="id('E.3.3')"/>
        <OMV name="c" xlink:href="id('E.3.2')"/>
        <OMV name="d" xlink:href="id('E.3.4')"/>
      </OMA>
    </OMA>
  </annotation-xml>
</semantics>
```

Here `OMA`, `OMS` and `OMV` are elements defined in the OpenMath standard for representing application, symbol and variable, respectively.

(Note that the application might or might not have a mechanism for extending DTDs. It will be the case, therefore that some applications will give well-formed, but not "valid," XML within `annotation-xml` elements. Consequently, some MathML applications using `annotation-xml` will not be validated. More flexibility is offered by the use of XML Schemas.)

5.4 Tools, Style Sheets and Macros for Combined Markup

The interaction of presentation and content markup can be greatly enhanced through the use of various tools. While the set of tools and standards for working with XML applications is rapidly evolving at the present, we can already outline some specific techniques.

In general, the interaction of content and presentation is handled via transformation rules on MathML trees. These transformation rules are sometimes called 'macros'. In principle, these rules can be expressed using any one of a number of mechanisms, including DSSSL, Java programs operating on a DOM, etc. We anticipate, however, that the principal mechanism for these transformations in most applications shall be XSLT.

In this section we discuss transformation rules for two specific purposes: for notational style sheets, and to simplify parallel markup.

5.4.1 Notational Style Sheets

Authors who make use of content markup may be required to deploy their documents in locales with notational conventions different than the default content rendering. It is therefore expected that transformation tools will be used to determine notations for content elements in different settings. Certain elements, e.g. `lambda`, `mean` and `transpose`, have widely varying common notations and will often require notational selection. Some examples of notational variations are given below.

- $\mathbf{V}$ versus $\vec{V}$
- $\tan x$ versus $\mathrm{tg}\, x$
- $\binom{n}{m}$ versus ${}_nC^m$ versus C_m^n versus C_n^m
- $a_0 + \frac{1|}{|a_1} + \ldots + \frac{1|}{|a_k}$ versus $[a_0, a_1, \ldots, a_k]$

Other elements, for example `plus` and `sin`, are less likely to require these features.

We observe that selection of notational style is sometimes necessary for correct understanding of documents by locale. For instance, the binomial coefficient C_m^n in French notation is equivalent to C_n^m in Russian notation.

A natural way for a MathML application to bind a particular notation to the set of content tags is with an XSLT style sheet [XSLT]. The examples of this section shall assume this is the mechanism to express style choices. (Other choices are equally possible, for example an application program may provide menus offering a number of rendering choices for all content tags.)

When writing XSLT style sheets for mathematical notation, some transformation rules can be purely local, while others will require multi-node context to determine the correct output notation. The following example gives a local transformation rule that could be included in a notational style sheet displaying open intervals as $]a,b[$ rather than as (a,b).

```
<xsl:template match="m:interval">
  <m:mrow>
    <xsl:choose>
      <xsl:when test="@closure='closed'">
        <m:mfenced open="[" close="]" separators=",">
          <xsl:apply-templates/>
        </m:mfenced>
      </xsl:when>
      <xsl:when test="@closure='open'">
        <m:mfenced open="]" close="[" separators=",">
          <xsl:apply-templates/>
        </m:mfenced>
      </xsl:when>
      <xsl:when test="@closure='open-closed'">
        <m:mfenced open="]" close="]" separators=",">
          <xsl:apply-templates/>
        </m:mfenced>
      </xsl:when>
      <xsl:when test="@closure='closed-open'">
        <m:mfenced open="[" close="[" separators=",">
          <xsl:apply-templates/>
        </m:mfenced>
      </xsl:when>
      <xsl:otherwise>
        <m:mfenced open="[" close="]" separators=",">
          <xsl:apply-templates/>
        </m:mfenced>
      </xsl:otherwise>
    </xsl:choose>
  </mrow>
</xsl:template>
```

Here `m` is established as the MathML namespace.

An example of a rule requiring context information would be:

```
<xsl:template match="m:apply[m:factorial]">
  <m:mrow>
    <xsl:choose>
      <xsl:when test="not(*[2]=m:ci) and not(*[2]=m:cn)">
        <m:mrow>
          <m:mo>(</m:mo>
          <xsl:apply-templates select="*[2]" />
          <m:mo>)</m:mo>
        </m:mrow>
      </xsl:when>
      <xsl:otherwise>
        <xsl:apply-templates select="*[2]" />
      </xsl:otherwise>
    </xsl:choose>
    <m:mo>!</m:mo>
  </m:mrow>
</xsl:template>
```

Other examples of context-dependent transformations would be, e.g. for the `apply` of a `plus` to render a-b+c, rather than a+ -b+c, or for the `apply` of a `power` to render $\sin^2 x$, rather than $\sin x^2$.

Notational variation will occur both for built-in content elements as well as extensions. Notational style for extensions can be handled as described above, with rules matching the names of any extension tags, and with the content handling (in a content-faithful style sheet) proceeding as described in Section 5.4.3.

5.4.2 Content-Faithful Transformations

There may be a temptation to view notational style sheets as a transformation from content markup to equivalent presentation markup. This viewpoint is explicitly discouraged, since information will be lost and content-oriented applications will not function properly.

We define a ‘content-faithful’ transformation to be a transformation that retains the original content in parallel markup (Section 5.3).

Tools that support MathML should be ‘content-faithful’, and not gratuitously convert content elements to presentation elements in their processing. Notational style sheets should be content-faithful whenever they may be used in interactive applications.

It is possible to write content-faithful style sheets in a number of ways. Top-level parallel markup can be achieved by incorporating the following rules in an XSLT style sheet:

```
<xsl:template match="m:math">
   <m:semantics>
      <xsl:apply-templates/>

      <m:annotation-xml m:encoding="MathML-Content">
         <xsl:copy-of select="."/>
      </annotation-xml>
   </m:semantics>
</xsl:template>
```

The notation would be generated by additional rules for producing presentation from content, such as those in Section 5.4.1. Fine-grained parallel markup can be achieved with additional rules treating `id` attributes.

5.4.3 Style Sheets for Extensions

The presentation tags of MathML form a closed vocabulary of notational structures, but are quite rich and can be used to express a rendering of most mathematical notations. Complex notations can be composed from the basic elements provided for presentation markup. In this sense, the presentation ability of MathML is open-ended. It is often useful, however, to give a name to new notational schemas if they are going to be used often. For example, we can shorten and clarify the ascending factorial example of Section 5.2.1, with a rule which replaces

```
<mx:a-factorial>X</mx:a-factorial>
```

with

```
<semantics>
   <apply> <factorial/> <mi>X</mi> </apply>
   <annotation-xml encoding="MathML-Presentation">
      <msup>
         <mn>1</mn>
         <mover accent="true">
            <mi>X</mi>
            <mo><mchar name="OverBar"/></mo>
         </mover>
      </msup>
   </annotation-xml>
</semantics>
```

Then the example would be more clearly written as:

```
<apply>
  <equivalent/>
  <ci>n</ci>
  <apply>
     <divide/>
     <mx:a-factorial><ci>n</ci></mx:a-factorial>
     <mx:a-factorial>
        <apply><minus/><ci>n</ci><cn>1</cn></apply>
     </mx:a-factorial>
   </apply>
</apply>
```

Likewise, the content tags form a fixed vocabulary of concepts covering the types of mathematics seen in most common applications. It is not reasonable to expect users to compose existing MathML content tags to construct new content concepts. (This approach is frought with technical difficulties even for professional mathematicians.) Instead, it is anticipated that applications whose mathematical content concepts extend beyond what is offered by MathML will use annotations within `semantics` elements, and that these annotations will use content description languages outside of MathML.

Often the naming of a notation and the identification of a new semantic concept are related. This allows a single transformation rule to capture both a presentation and a content markup for an expression. This is one of the areas of MathML that benefits most strongly from the use of macro processing.

```
<mx:rank/>
```

and

```
<mx:tr>X</mx:tr>
```

and respectively transform them to

```
<semantics>
   <ci><mo>rank</mo></ci>
   <annotation-xml encoding="OpenMath">
      <OMS cd="linalg1" name="rank"/>
   </annotation-xml>
</semantics>
```

and

```
<apply>
 <transpose/>
 <ci>X</ci>
</apply>
```

The lengthy sample encoding of rank($u^T v$)=1, from Section 5.2.1 could then be condensed to

```
<apply>
  <eq/>
  <apply>
    <mx:rank/>
    <apply> <times/> <mx:tr>u</mx:tr> <ci>v</ci> </apply>
   </apply>
  <cn>1</cn>
</apply>
```

From this example we see how the combination of presentation and content markup could become much simpler and effective to generate as standard style-sheet libraries become available.

Chapter 6

Characters, Entities and Fonts

6.1 Introduction

6.1.1 The Intent of Character Names

Notation and symbols have proved very important for mathematics. Mathematics has grown in part because of the succinctness and suggestiveness of its evolving notation. There have been many new signs evolved for use in mathematical notation, and mathematicians have not held back from making use of many symbols originally developed elsewhere. The result is that mathematics makes use of a very large collection of symbols. It is difficult to write mathematics fluently if these characters are not available for use in coding. It is difficult to read mathematics if corresponding glyphs are not available for presentation on specific display devices.

This situation posed a problem for the first W3C Math Working Group when it was brought into existence. It did not fall naturally within the purview of a developing a specification enabling mathematics to be used with HTML and producing a DTD for this to worry about more than the entities allowed in the DTD. However, as experience has shown, a long list of entities with no means to display them is of little use, and a cause of frequent frustrations in trying use a standard. On the other hand, a large collection of glyphs and fonts of characters without a standard way to refer to them is not of much use either.

The W3C Math Working Group has therefore took on directly the task of specification of part of the full mechanism of needed to proceed from notation to final presentation, and started collaboration with organizations undertaking specification of the rest.

For instance, in MathML 1 we tried to use entity names for the many character signs that are contained in ISO TR 9573, which supersedes the ISO TR 8879 annex as far as mathematics is concerned. There are considerations of mathematical usage that do on occasion militate against this, and the TR 9573 lists need supplementing. There was the hope of agreeing with the TR 9573 WG on suitable extensions, in the course of the revision of their document that they were undertaking. That has not actually happened, and the expected TR 9573 revision has not appeared either.

The STIX project of the STIPUB group of scientific and technical publishers has also been working since 1997 toward a common collection of mathematical symbols and names. The W3C Math Working Group itself has collaborated with that project and expects to have to issue further updates on the matter of character entities as a consequence of useful work of this project and others. For the latest character tables and fonts information, see the W3C Math Working Group home page.

6.1.2 The STIX Project

The first STIX project team leader, Nico Poppelier, is a member of the W3C Math Working Group. The STIX project, set up by the STIPUB group of publishers includes the American Chemical Society (ACS), the American Institute of Physics (APS), the American Mathematical Society (AMS), the AMerican Physical Society (APS), Elsevier Science Publishers, the Institute of Electrical and Electronic Engineers (IEEE). An initial aim was to formulate a collection of characters needed in the course of scientific and technical publishing. A database of characters in common use has been produced by collaborating publishing organizations, including information from the TEX world, Springer Verlag (Heidelberg), Design Science Inc., Wolfram Research Inc., the Association for Computing Machinery (ACM) in addition to the above-mentioned. The coordination and the major portion of the work on this have been carried out by Barbara Beeton of the AMS.

The STIX team has proposed to the Unicode Technical Committee (UTC) of the Unicode consortium the additions to the next revision of the Unicode character set that this process shows are needed, together with the appropriate character codes. This has been the subject of on-going negotiation for some time. In March 2000 a honed proposal supported by the UTC went on the the ISO WG2 meeting in Beijing which deals with incorporation of new material into the standard ISO 10646. The final results of that deliberation, which it is hoped will confirm assignement of code-points put forward by the UTC will be incorporated into the information made public by the Math WG.

Finally, the STIX project's intention has always been to commission the production of a complete set of fonts covering those Unicode characters for science and technology, to be made available to the public under license, but free of charge. The STIPUB group recognizes that easy availability of the characters and fonts greatly facilitates communication and publication. At the start of the year 2000 the process of commisioning the making of fonts is underway, and their wide-spread availability is hoped with one or two years.

6.1.3 Character Listings

This chapter of the MathML Specification contains a listing of character names for use in MathML.

To provide more background on the characters used by mathematics we have used a large comparative database showing codes and meanings in other common math environments. The W3C Math Working Group is very grateful to Elsevier Science, to Wolfram Research (makers of Mathematica ®) and to Design Science (Makers of MathType ®) for making available to us so much useful data.

In MathML 1 the characters of the mathematical sciences were listed as entities. This is coherent with thinking in terms of SGML markup and the use of DTDs. For the XML world with its use of documents well-formedness is to be sufficient for the examination of a particular one, which does not require validation against a DTD, where character entities would be found declared. The next development that is expected to replace the DTD as a specifier of a class of documents is that of Schemas. The specification for Schemas is presently under active development at the W3C. Though the final form of Schemas is not yet clear, it is known that their use precludes effective use of large lists of entities. For that reason MathML 2 passes from the use of entities to name mathematical characters, which becomes a deprecated usage, to the use of `Error: mchar` elements. For this reason the tables below just list the suggested character names, which should be used in the form `<mchar name="character_name" />`.

6.1.4 Non-Marking Characters

Some characters although important for the quality of print rendering do not directly have glyph marks that correspond. They are called here non-marking characters. Below we have a table of those adopted for the purposes of MathML. Their roles are discussed in Chapter 3 and Chapter 4, respectively. The values of the spaces given are recommendations. Some of these characters do not have official Unicode values, and some are given as combinations of Unicode characters employing the special mathematics modifier character (U02063). The correspondence between the spacing values mentioned below and those in the Unicode descriptions are not exact, but are good matches.

It used to be in MathML 1.0 that there were a number more non-marking character entities listed here. These were conerned with composition control, such as line-breaking, In MathML 2 such control is effected by the use of the proper attributes on the `mspace` element.

Character name	Unicode	Description
	00009	tabulator stop; horizontal tabulation
0000A	force a line break; line feed
&Space;	00020	one em of space in the current font
	000A0	space that is not a legal breakpoint
	0200B	space of no width at all
	0200A	space of width 1/18 em
	02009	space of width 3/18 em
	02005	space of width 4/18 em
	02005-0200A	space of width 5/18 em
	0200A-02063	space of width -1/18 em
	02009-02063	space of width -3/18 em
	0205F-02063	space of width -4/18 em
	02005-02063	space of width -5/18 em
⁢	02062	marks multiplication when it is understood without a mark (Section 3.2.4
⁡	02061	character showing function application in presentation tagging (Section 3.2.4

6.1.5 Printing Character Symbol Listings

Even though the situation concerning availability of character codes from Unicode and under ISO 10646 is not yet fully clear at the time of writing, we have decided to proceed on the assumption that the code points suggested to ISO WG2 by the UTC will be confirmed. As before we can only reiterate that for current developments on details of character standards as far as they influence mathematical formalism the Home Page of the W3C Math WG should be consulted.

The Math WG started from the ISO 9573-13 proposal, as conveyed to us from Anders Berglund, and added a number of informative additional aliases based in the practice of the mathematical typesetting community. The main influence outside ISO has been the names to be found in the TEX community because they inform the practice of the contributors to the STIX character database mentioned above.

To facilitate comprehension of a fairly large list of names, which totals over 2000 in this case, we offer the same information in more than one form.

We have characters listed by name and sample glyphs for all of them. Each character name is accompanied by a code for a character grouping chosen from a list given below, a short verbal description, and a Unicode hex code if there is a corresponding sample glyph to be found in ISO 10646, now extended in accordance with the proposal forwarded by the UTC to ISO WG2 in March 2000. We have excluded, with very few exceptions that seemed to us compelling, other characters that may have appeared in the corresponding lists in MathML 1. Those characters thus *lost* will be found to be used very infrequently in the experience of mathematical publishers, or simply to be completley unacceptable for inclusion in Unicode. However MathML 2 does provide the `mglyph` and `csymbol` elements to accommodate new characters that authors may wish to introduce.

The character listings by alphabetical and Unicode order in Section 6.1.7 have now been brought more into line with the corresponding ISO character sets than was the case in MathML 1.0, in that if some part of a set is included then the entire set is included. In addition, the group ISOCHEM has been dropped as more properly the concern of chemists. These changes have also been reflected in the entity declarations in the DTD in Appendix A.

6.1.6 Special Constants

To commence we list separately a few of the special characters which MathML has seen fit to be a little radical in introducing. These have been accorded new Unicode values. There used also the be entries below for `&true;`, `&false;` and `&NotANumber;`, but these do not yet have Unicode points assigned to them so have been removed. They can be reintroduced by the character extension mechanisms provided by the `mchar` and `csymbol` elements.

Entity name	Unicode	Description
ⅅ	02145	D for use in differentials, e.g. within integrals
ⅆ	02146	d for use in differentials, e.g. within integrals
ⅇ	02147	e for use for the exponential base of the natural logarithms
ⅈ	02148F	i for use as a square root of -1

6.1.7 Alphabetical Lists

The first table offered is a very large ASCII listing of printing entity names, ordered alphabetically, with upper-case preceding lower-case as in ASCII order. There is also an ASCII listing of printing characters ordered by Unicode number. The Unicode point points are those of the current proposal which will, it is expected eventaully be part of the next revision, Unicode 4. Unicode 3 has just been published in February 2000. Next we have collections of the entities in entity sets which correspond to the groupings in the corresponding ISO documents.

6.1.8 ISO Character Set Groupings

In addition, we list the above material in the groupings used by ISO 9573-13 introduced. This table makes explicit the entity groupings and provides links to ASCII listings of the groups and HTML tabular listings which display the glyphs, as well.

6.1.8.1 ISO Symbol Sets

The symbols for mathematics that ISO have considered are organized, for both historical and mnemonic reasons into groupings with somewhat descriptive names. In the tables below we reproduce the newly proposed versions of these groups and give the corresponding Unicode sample glyphs. The entries are organized alphabetically by character name.

It should be noted that the sample glyphs given here are in GIF files intended for viewing on a monitor's screen at 72dpi. They are not suitable for printing, and in particular do not constitute a set of fonts covering the symbols of mathematics. Such a set of fonts is under development in more than one context. The MathML Working Group is engaged in the effort of ensuring that such fonts will be readily publicly available.

This first block of sets includes mostly non-letter symbols, along with a few letters loaded with mathematical semantics.

Group	Descriptive Name
ISOAMSA	Added Math Symbols: Arrows
ISOAMSB	Added Math Symbols: Binary Operators
ISOAMSC	Added Math Symbols: Delimiters
ISOAMSN	Added Math Symbols: Negated Relations
ISOAMSO	Added Math Symbols: Ordinary
ISOAMSR	Added Math Symbols: Relations
ISOTECH	General Technical
ISOPUB	Publishing
ISODIA	Diacritical Marks
ISONUM	Numeric and Special Graphic
ISOBOX	Box and Line Drawing

6.1.8.2 ISO Character Sets for Mathematics Alphabets

Mathematical literature displays the common use of particular font styles. Characters representing given letters which differ only in the glyph presentation are in principle not different for the purposes of a character registry such as Unicode, which is not supposed to take into account mere font differences. However usage has meant that both ISO and Unicode, like mathematics, recognize them as different entities. Therefore we wish to include lists for Greek, script, open face (also known as double struck or blackboard bold), and fraktur (also known as gothic or German) fonts. The UTC has accepted a proposal for the inclusion of alphabetic character runs in Unicode Plane 1 for the express use of mathematics, brought to them by Murray Sargent of Microsoft and supported by the STIX Project as a compromise solution. However the tenets of the UTC preclude the duplication, if at all possible, of methods for encoding a character which conventionally has esentially one glyphic representation. Thus there are *holes* at certain points in the alphabetic runs for mathematical use in Plane 1 coding. These holes will, however, be reserved and not used for anything else, and so can be used, internally, in the obvious way by an application handling mathematics.

Group	Descriptive Name
ISOGRK3	Greek Symbols
ISOMSCR	Math Alphabet Script
ISOMOPF	Math Alphabet Open Face
ISOMFRK	Math Alphabet Fraktur

Chapter 7

The MathML Interface

To be effective, MathML must work well with a wide variety of renderers, processors, translators and editors. This chapter addresses some of the interface issues involved in generating and rendering MathML. Since MathML exists primarily to encode mathematics in Web documents, perhaps the most important interface issues are related to embedding MathML in HTML [HTML4.0] and XHTML [XHTML1.0] and [XHTML1.1].

There are three kinds of interface issues that arise in embedding MathML in other XML documents. First, MathML must be semantically integrated. MathML markup must be recognized as valid embedded XML content, and not as an error. This is primarily a question of managing namespaces in XML [Namespaces].

Second, in the case of HTML/XHTML, MathML rendering must be integrated into browser software. Some browsers already implement MathML rendering natively, and one can expect more browsers will do so in the future. At the same time, other browsers have developed infrastructure to facilitate the rendering of MathML and other embedded XML content by third-party software. Using these browser specific mechanisms generally requires some additional interface markup of some sort to activate.

Third, other tools for generating and processing MathML must be able to intercommunicate. A number of MathML tools have been or are being developed, including editors, translators, computer algebra systems, and other scientific software. However, since MathML expressions tend to be lengthy, and prone to error when entered by hand, special emphasis must be given to insuring that MathML can be easily generated by user-friendly conversion and authoring tools, and that these tools work together in a dependable, platform and vendor independent way.

The W3C Math working group is committed to providing support to software vendors developing all kinds of MathML tools. The working group monitors the public mailing list www-math@w3.org, and will attempt to answer questions about the MathML specification. The working group works with MathML developer and user groups. For current information about MathML tools, applications and user support activities, consult the home page of the W3C Math Working Group.

7.1 Embedding MathML in other Documents

While MathML can be used in isolation as a language for exchanging mathematical expressions between MathML-aware applications, the primary anticipated use of MathML is to encode mathematical expression within larger documents. MathML is ideal for embedding math expressions in other applications of XML.

In particular, we focus here on the mechanics of embedding MathML in XHTML. XHTML [XHTML1.0] is a recently released W3C Recommendation formulating a family of current and future XML-based document types and modules that reproduce, subset, and extend HTML 4. While HTML 4 [HTML4.0] is the dominant language of the Web today, one may anticipate a shift from HTML 4 to XHTML. Indeed, XHTML already renders properly in HTML 4 user agents.

Since MathML and XHTML share a common XML framework, namespaces provide a standard mechanism for embedding MathML in XHTML. While some popular user agents also support inclusion of MathML directly in HTML 4 as "XML data islands," the view point we adopt here is that this is a transitional strategy, and we don’t elaborate on it. Consult your user agent documentation for specific information on its support for embedding XML in HTML.

7.1.1 MathML and Namespaces

Embedding MathML in XML-based documents in general, and XHTML in particular, is a matter of managing namespaces. See the W3C Recommendation "Namespaces in XML" [Namespaces] for full details.

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

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

Using namespaces, embedding a MathML expression in a larger XML document is merely a matter of identifying the MathML markup as residing in the MathML namespace. This can be accomplished by either explicitly identifying each MathML element name by attaching a namespace prefix, or by declaring a default namespace on an enclosing element.

To declare a namespace, one uses an `xmlns` attribute, or an attribute with an `xmlns` prefix. When the `xmlns` attribute is used alone, it sets the namespace for the element on which it appears, and for any children elements.

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.

Example:

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

These two methods of namespace declaration can be used together. For example, by using both an explicit document-wide namespace prefix, and default namespace declarations on individual math elements, it is possible to localize namespace related markup to the top-level `math'` element. This is also important for implementation with some user agents, since attaching rendering behaviors to element currently requires an explicit namespace prefix in these browsers. At the same time, a number of MathML authoring tools are not yet namespace-aware, and thus the ability to use markup without prefixes is also desirable in the short term.

Example:

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

7.1.1.1 Document Validation Issues

The use of namespace prefixes creates an issue for DTD validation of documents embedding MathML. DTD validation requires knowing the literal (possibly prefixed) element names used in the document. However, the Namespaces in XML Recommendation [Namespaces] allows the prefix to be changed at arbitrary points in the document, since namespace prefixes may be declared on any element.

The ‘historical’ method of bridging this gap was to write a DTD with a fixed prefix, or in the case of XHTML and MathML, with no prefix, and mandate that the specified form must be used throughout the document. However, this is somewhat restricting for a modular DTD that is intended for use in conjunction with another DTD, which is exactly the situation with MathML in XHTML. In essence, the MathML DTD would have to ‘allocate’ a prefix for itself and hope no other module uses the same prefix to avoid name clashes, thus losing one of the main benefits of XML namespaces.

One strategy for addressing this problem is to make every element name in the DTD be accessed by an entity reference. This means that by declaring a couple of entities to specify the prefix before the DTD is loaded, the prefix can be chosen by a document author, and compound DTDs that include several modules can, without changing the module DTDs, specify unique prefixes for each module to avoid clashes. The MathML DTD has been designed in this fashion. See Section A.1, [Modularization] and [Building] for details.

An extra issue arises in the case where explicit prefixes are used on the top-level math element, but a default namespace is used for other MathML elements. In this case, one wants the MathML module to be included into XHTML with the prefix set to empty. However, the ‘driver’ DTD file that sets up the inclusion of the MathML module would then need to define a new element called m:math. This would allow the top-level math element to use an explicit prefix, for attaching rendering behaviors in current browsers, while the contents would not need an explicit prefix, for ease of interoperability between authoring tools, etc.

7.1.1.2 Compatibility Suggestions

While the use of namespaces to embed MathML in other XML applications is completely described by the relevant W3C recommendations, a certain degree of pragmatism is still called for at present. Support for XML, namespaces and rendering behaviors in popular user agents is not always fully in alignment with W3C Recommendations. In some cases the software predates the relevant standard, and in other cases, the relevant standards are not yet complete.

During the transitional period in which some software may not be fully namespace-aware, a few conventional practices will ease compatibility problems. After surveying a number of user agents and other MathML-aware software applications, we offer the following suggestions.

1. If you use namespace prefixes with MathML markup, use m: as a conventional prefix for the MathML namespace. Using an explicit prefix is probably safer for compatibility in current user agents.
2. If you use namespace prefixes, pick one and use it consistently within a document.
3. Explicitly declare the MathML namespace on all `math` elements.

Examples.

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

Or

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

Note that these suggestions alone may not be sufficient for creating functional Web pages containing MathML markup. It will generally be the case that some additional document-wide markup will be required. Additional work may also be required to make all MathML instances in a document compatible with document-wide declarations. This is particularly true when documents are created by cutting and pasting MathML expressions, since current tools will probably not be able to query global namespace information.

Consult the W3C Math Working Group homepage for compatibility and implementation suggestions for current browsers and other MathML-aware tools.

7.1.2 The Top-Level math Element

MathML specifies a single top-level or root math element, which encapsulates each instance of MathML markup within a document. All other MathML content must be contained in a math element; equivalently, every valid, complete MathML expression must be contained in <math> tags. The math element must always be the outermost element in a MathML expression; it is an error for one math element to contain another.

Applications that return sub-expressions of other MathML expressions, for example as the result of a cut-and-paste operation, should always wrap them in <math> tags. Ideally, the presence of enclosing <math> tags should be a very good heuristic test for MathML content. Similarly, applications which insert MathML expressions in other MathML expressions must take care to remove the <math> tags from the inner expressions.

The math element can contain an arbitrary number of children schemata. The children schemata render by default as if they were contained in a mrow element.

The attributes of the math element are:

class, id, style Provided for style sheet and DOM compatibility.

macros This attribute provides a way of pointing to external macro definition files. Macros are not part of the MathML specification, and much of the functionality provided by macros in MathML can be accommodated by XSL transformations [XSLT]. However, the macros attribute is provided to make possible future development of more streamlined, MathML-specific macro mechanisms. The value of this attribute is a sequence of URLs or URIs, separated by whitespace

mode The mode attribute specifies whether the enclosed MathML expression should be rendered in a display style or an in-line style. Allowed values are display and inline (default). This attribute is deprecated in favor of the new display attribute, or the standard CSS2 'display' property with the analogous block and inline values.

display The display attribute replaces the deprecated mode element. It specifies whether the enclosed MathML expression should be rendered in a display style or an in-line style. Allowed values are block and inline (default).

The attributes of the math element affect the entire enclosed expression. It is, in a sense, 'inward looking'. However, to render MathML properly in a browser, and to integrate it properly into an XHTML document, a second collection of 'outward looking' attributes are also useful.

While general mechanisms for attaching rendering behaviors to elements in XML documents are under development, wide variations in strategy and level of implementation remain between various existing user agents. Consequently, the remainder of this section describes attributes and functionality that are desirable for integrating third-party rendering modules with user agents:

overflow In cases where size negotiation is not possible or fails (for example in the case of an extremely long equation), this attribute is provided to suggest an alternative processing method to the renderer. Allowed values are

- **scroll** The window provides a viewport into the larger complete display of the mathematical expression. Horizontal or vertical scrollbars 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.
- **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.

altimg This attribute provides a graceful fall-back for browsers that do not support embedded elements. The value of the attribute is an URL.

alttext This attribute provides a graceful fall-back for browsers that do not support embedded elements or images. The value of the attribute is a text string.

7.1.3 Invoking MathML Processors

In browsers where MathML is not natively supported, we anticipate that MathML rendering will be carried out via embedded objects such as plug-ins, applets, or helper applications. The direction which has begun emerging for invoking third-party rendering and processing software is elucidated in the W3C Working Draft "Behavioral Extensions to CSS" [Behaviors].

Behavioral extensions use the linking mechanism of CSS to attach executable components to elements. Typically, the executable components involve script code which manipulate the DOM to instantiate other MathML processing components. Using experimental implementations of behavior extensions in current user agents, it is possible to attach processing components to math elements which use applets or plug-ins to render MathML markup in an XHTML page.

Work on on Behavior Extensions to CSS is ongoing at W3C, and existing implementations must be regarded as experimental at the time. However, it offers a very promising direction for powerful and flexible invocation of third-party MathML processors.

MIME types offer an alternative strategy that can also be used in current user agents to invoke a MathML renderer. This is primarily useful when referencing separate files containing MathML markup from an `EMBED` or `OBJECT` element. The W3C Math Working Group suggests that generic MathML be assigned the MIME type `text/x-mathml`, and for browser registry, we suggest the standard file extension `.mml` be used. In MathML 1.0, `text/mathml` was given as the suggested MIME type. However, the assignment of MIME types to XML applications has come into question in the interim. Thus, beginning with MathML 2.0, we suggest instead using the less-regulated experimental MIME type `text/x-mathml`.

Although rendering MathML expressions typically occurs in place in a Web browser, other MathML processing functions take place more naturally in other applications. Particularly common tasks include opening a MathML expression in an equation editor or computer algebra system.

At present, there is no standard way of selecting between various applications which might be used to render or process embedded MathML. As work progresses on coordination between browsers and embedded elements and the Document Object Model [DOM], providing this kind of functionality should be a priority. Both authors and readers should be able to indicate a preference about what MathML application to use in a given context. For example, one might imagine that some mouse gesture over a MathML expression causes a browser to present the reader with a pop-up menu, showing the various kinds of MathML processing available on the system, and the MathML processors recommended by the author.

Since MathML is most often generated by authoring tools, it is particularly important that opening a MathML expression in an editor should be easy to do and to implement. In many cases, it will be desirable for an authoring tool to record some information about its internal state along with a MathML expression, so that an author can pick up editing where he or she left off. The MathML specification does not explicitly contain provisions for recording information about the authoring tool. In some circumstances, it may be possible to include authoring tool information that applies to an entire document in the form of meta-data; interested readers are encouraged to consult the W3C Metadata Activity for current information about metadata and resource definition. For encoding authoring tool state information that applies to a particular MathML instance, readers are referred to the possible use of the `semantics` element for this purpose.

In the short term, regardless of the methodology, implementors of embedded MathML processing applications are encouraged to try to allow for the following kinds of functionality:

- An author wishing to reach an audience as wide as possible might want MathML to be rendered by any available processor.
- An author targeting a specific audience might want to indicate that a particular MathML processor be used.
- A reader might wish to specify which of several available processors installed locally should be used.

7.1.4 Mixing and Linking MathML and HTML

In order to be fully integrated into XHTML, it should be possible not only to embed MathML in XHTML, but also to embed XHTML in MathML. However, the problem of supporting XHTML in MathML presents many difficulties. Moreover, the problems are not specific to MathML; they are problems for XML applications in XHTML generally. Therefore, at present, the MathML specification does not permit any XHTML elements within a MathML expression, although this may be subject to change in a future revision of MathML.

In most cases, XHTML elements either do not apply in mathematical contexts (headings, paragraphs, lists, etc.), or MathML already provides equivalent or better functionality specifically tailored to mathematical content (tables, style changes, etc.). However, there are two notable exceptions.

7.1.4.1 Linking

MathML has no element that corresponds to the XHTML anchor element a. In XHTML, anchors are used both to make links, and to provide locations to which a link can be made. MathML, as an XML application, defines links by the use of the mechanism described in the W3C Working Draft "XML Linking Language" [XLink]. The reader is cautioned that this is at present still a working draft, and is therefore subject to future revision. Since the MathML linking mechanism is defined in terms of the XML linking specification, the same proviso holds for it as well.

A MathML element is designated as a link by the presence of the attribute `xlink:href`. To use the attribute `xlink:href`, it is also necessary to declare the appropriate namespace. Thus, a typical MathML link might look like:

```
<mrow xmlns:xlink="http://www.w3.org/1999/xlink"
      xlink:href="sample.xml">
  ...
</mrow>
```

MathML designates that almost all elements can be used as XML linking elements. The only elements that cannot serve as linking elements are those such as the `sep` element, which exist primarily to disambiguate other MathML constructs and in general do not correspond to any part of a typical visual rendering. The full list of exceptional elements that cannot be used as linking elements is given in the table below.

Table 7.1: MathML elements that cannot be linking elements.

`mprescripts`	`none`	`sep`
`malignmark`	`maligngroup`	

Note that the XML Linking [XLink] and XML Pointer Language [XPointer] specifications also define how to link *into* a MathML expressions. Be aware, however, that such links may or may not be properly interpreted in current software.

7.1.4.2 Images

The `IMG` element has no MathML equivalent. The decision to omit a general mechanism for image inclusion from MathML was based on several factors. However, the main reason for not providing an image facility is that MathML takes great pains to make the notational structure and mathematical content it encodes easily available to processors, whereas information contained in images is only available to a human reader looking at a visual representation. Thus, for example, in the MathML paradigm, it would be preferable to introduce new glyphs via the `mglyph` element which at a minimum identifies them as glyphs, rather than simply including them as images.

Finally, apart from the introduction of new glyphs, many of the situations where one might be inclined to use an image amount to some sort of labeled diagram. For example, knot diagrams, Venn diagrams, Dynkin diagrams, Feynman diagrams and complicated commutative diagrams all fall into this category. As such, their content would be better encoded via some combination of structured graphics and MathML markup. Because of the generality of the 'labeled diagram' construction, the definition of a markup language to encode such constructions extends beyond the scope of the current W3C Math activity. (See http://www.w3.org/Graphics for further W3C activity in this area.)

7.2 Generating, Processing and Rendering MathML

Information is increasingly 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. Thus, although MathML can be written by hand and read by humans, the future of MathML is also tied to the ability to process it with software tools.

There are many different kinds of MathML editors, translators, processors and renderers. What it means to support MathML varies widely between applications. For example, the issues that arise with a MathML-compliant validating parser are very different from those for a MathML-compliant equation editor.

In this section, guidelines are given for describing different types of MathML support, and for quantifying 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 and interoperability between MathML applications by accurately characterizing their capabilities in quantifiable terms.

7.2.1 MathML Compliance

A valid MathML expression is an XML construct determined by the MathML DTD together with the additional requirements given in the specifications of the MathML document.

We define a 'MathML processor' to mean any application that can accept, produce, or 'roundtrip' a valid MathML expression. An example of an application that might round-trip a MathML expression might be an editor that writes a new file even though no modifications are made.

We specify three forms of MathML compliance:

1. A MathML-input-compliant processor must accept all valid MathML expressions, and faithfully translate all MathML expressions into application-specific form allowing native application operations to be performed.
2. A MathML-output-compliant processor must generate valid MathML, faithfully representing all application-specific data.
3. A MathML-roundtrip-compliant 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 DTD and specification) under any 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 suggested rendering rules included in Chapter 3. However, in general, developers are given wide latitude in interpreting what kind of MathML implementation is meaningful for their own particular application.

To clarify the difference between compliance and interpretation of what is meaningful, consider some examples:

1. In order to be MathML-input-compliant, 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 included those containing presentation markup, it would be technically correct to characterize the application as MathML-input-compliant.
3. A 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-compliant.

As the previous examples show, to be useful, the concept of MathML compliance 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 compliance. 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 of MathML expressions at http://www.w3.org/Math/testsuite.

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 is possible, for example, to characterize quantitatively the hypothetical computer algebra interface mentioned above by saying that it is a MathML-input compliant processor which meaningfully implements MathML content markup, including all of the expressions given under http://www.w3.org/Math/testsuite/tests/4.

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-compliant processors, there is also a MathML validator online at http://www.w3.org/Math/validator. Developers of MathML-output-compliant 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.

7.2.1.1 Deprecated MathML 1.x Features

MathML 2.0 contains a number of MathML 1.x constructs which are now deprecated. We now clarify the relation between deprecated features and MathML 2.0 compliance.

1. In order to be MathML-output-compliant, authoring tools may not generate MathML markup containing deprecated features.
2. In order to be MathML-input-compliant, rendering/reading tools must support deprecated features if they are to be MathML 1.x compliant. They do not have to support deprecated features to be considered MathML 2.0 compliant. However, all tools are encouraged to support the old forms as much as possible.
3. In order to be MathML-roundtrip-compliant, a processor need only preserve MathML equivalence on expressions containing no deprecated features.

7.2.2 Handling of Errors

If a MathML-input-compliant 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-compliant 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.

7.2.3 Attributes for unspecified data

The MathML attributes described in the MathML specification are necessary for presentation and content 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. Although it can be done using extensions of the standard DTD, some authors will wish to use non-standard attributes to take advantage of renderer-specific capabilities while remaining strictly in compliance with the standard DTD.

To allow this, the MathML 1.0 specification allowed the attribute `other` on all elements, for use as a hook to pass on renderer-specific information. In particular, it was intended as a hook for passing information to audio renderers, computer algebra systems, and for pattern matching in future macro/extension mechanisms. The motivation for this approach to the problem was historical, looking to PostScript, for example, where comments are widely used to pass information that is not part of PostScript.

In the mean time, however, the development of a general XML namespace mechanism has made 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.

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 3.6.1), that information should be passed in using the `other` attribute:

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

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

```
<body xmlns:my="http://www.myrenderer.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 loop-hole for circumventing the core conventions for MathML markup. We trust both authors and applications will use non-standard attributes judiciously.

7.3 Future Extensions

If MathML is to remain useful in the future, it is to be expected that MathML will need to be extended and revised in various ways. Some of these extensions can be easily foreseen; for example, as work on behavioral extensions to CSS proceeds, MathML will likely need to be extended as well.

Similarly, there are several kinds of functionality that are fairly obvious candidates for future MathML extensions. These include macros, style sheets, and perhaps a general facility for ‘labeled diagrams’. However, there will no doubt be other desirable extensions to MathML that will only emerge as MathML is widely used. For these extensions, the W3C Math working group relies on the extensible architecture of XML, and the common sense of the larger Web community.

7.3.1 Macros and Style Sheets

The development of style-sheet mechanisms for XML is part of the ongoing XML activity of the World Wide Web Consortium. Both XSL and CSS are working to incorporate greater support for mathematics.

In particular, XSL Transformations [XSLT] are likely to have a large impact on the future development of MathML. Macros has traditionally contributed greatly the usability and effectiveness of mathematics encodings. Further work developing applications of XSLT tailored specifically to MathML is clearly called for.

Some of the possible uses of macro capabilities for MathML include:

Abbreviation One common use of macros is for abbreviation. Authors needing to repeat some complicated but constant notation can define a macro. This greatly facilitates hand authoring. Macros that allow for substitution of parameters facilitate such usage even further.

Extension of Content Markup By defining macros for semantic objects, for example a binomial coefficient, or a Bessel function, one can in effect extend the content markup for MathML. Such a macro could include an explicit semantic binding, or such a binding could be easily added by an external applications. Narrowly defined disciplines should be able to easily introduce standardized content markup by using standard macro packages. For example, the OpenMath project could release macro packages for attaching OpenMath content markup.

Rendering and Style Control Another basic way in which macros are often used is to provide a way of controlling style and rendering behavior by replacing high-level macro definitions. This is especially important for controlling the rendering behavior of MathML content tags in a context sensitive way. Such a macro capability is also necessary to provide a way of attaching renderings to user-defined XML extensions to the MathML core.

Accessibility Reader-controlled style sheets are important in providing accessibility to MathML. For example, a reader listening to a voice renderer might, by default, hear a bit of MathML presentation markup read as ‘D sub x sup 2 of f’. Knowing the context to be multi-variable calculus, the reader may wish to use a style sheet or macro package that instructs the renderer to render this `<msubsup>` element as ‘second derivative with respect to x of f’.

7.3.2 XML Extensions to MathML

The set of elements and attributes specified in the MathML specification are necessary for rendering common mathematical expressions. It is recognized that not all mathematical notation is covered by this set of elements, that new notations are continually invented, and that sub-communities within mathematics often have specialized notations; and furthermore that the explicit extension of a standard is a necessarily slow and conservative process. This implies that the MathML standard could never explicitly cover all the presentational forms used by every sub-community of authors and readers of mathematics, much less encode all mathematical content.

In order to facilitate the use of MathML by the widest possible audience, and to enable its smooth evolution to encompass more notational forms and more mathematical content (perhaps eventually covered by explicit extensions to the standard), the set of tags and attributes is open-ended, in the sense described in this section.

MathML is described by an XML DTD, which necessarily limits the elements and attributes to those occurring in the DTD. Renderers desiring to accept non-standard elements or attributes, and authors desiring to include these in documents, should accept or produce documents that conform to an appropriately extended XML DTD that has the standard MathML DTD as a subset.

MathML-compliant renderers are allowed, but not required, to accept non-standard elements and attributes, and to render them in any way. If a renderer does not accept some or all non-standard tags, it is encouraged either to handle them as errors as described above for elements with the wrong number of arguments, or to render their arguments as if they were arguments to an `mrow`, in either case rendering all standard parts of the input in the normal way.

Chapter 8

Document Object Model for MathML

8.1 Introduction

This document extends the Core API of the DOM Level 1 to describe objects and methods specific to MathML elements in documents. The functionality needed to manipulate basic hierarchical document structures, elements, and attributes will be found in the core document; functionality that depends on the specific elements defined in MathML will be found in this document.

The actual DOM specification appears in Appendix E.

The goals of the MathML-specific DOM API are:

- To specialize and add functionality that relates specifically to MathML elements.
- To provide convenience mechanisms, where appropriate, for common and frequent operations on MathML elements.

This document includes the following specializations for MathML:

- A `MathMLElement` interface derived from the core interface `Element`. MathMLElement specifies the operations and queries that can be made on any MathML element. Methods on `MathMLElement` include those for the retrieval and modification of attributes that apply to all MathML elements.
- Various specializations of `MathMLElement` to encode syntactical restrictions imposed by MathML.
- Specializations of `MathMLElement` representing all MathML elements with attributes extending beyond those specified in the `MathMLElement` interface. For all such attributes, the derived interface for the element contains explicit methods for setting and getting the values.
- Special methods for insertion and retrieval of children of MathML elements. While the basic methods inherited from the `Node` and `Element` interfaces must clearly remain available, it is felt that in many cases they may be misleading. Thus, for instance, the `MathMLFractionElement` interface provides for access to `numerator` and `denominator` attributes; a call to `setDenominator(newNode)` is less ambiguous from a calling application's perspective than a call to `Node::replaceNode(newNode, Node::childNodes().item(2))`.

MathML specifies rules that are invisible to generic XML processors and validators. The fact that MathML DOM objects are required to respect these rules, and to throw exceptions when those rules are violated, is an important reason for providing a MathML-specific DOM extension.

There are basically two kinds of additional MathML grammar and syntax rules. One kind involves placing additional criteria on attribute values. For example, it is not possible in pure XML to require that an attribute value be a positive integer. The second kind of rule specifies more detailed restrictions on the child elements (for example on ordering) than are given in the DTD. For example, it is not possible in XML to specify that the first child be interpreted one way, and the second in another. The MathML DOM objects are required to provide this interpretation.

MathML ignores whitespace occurring outside token elements. Non-whitespace characters are not allowed there. Whitespace occurring within the content of token elements is 'trimmed' from the ends (i.e. all whitespace at the beginning and end of the content is removed), and 'collapsed' internally (i.e. each sequence of 1 or more whitespace characters is replaced with one blank character). The MathML DOM elements perform this whitespace trimming as necessary. In MathML, as in XML, 'whitespace' means blanks, tabs, newlines, or carriage returns, i.e. characters with hexadecimal Unicode codes U+0020, U+0009, U+000a, or U+000d, respectively.

8.1.1 MathML DOM Extensions

It is expected that a future version of the MathML DOM may deal with issues which are not resolved here. Some of these are described here.

8.1.1.1 Style Issues and Implied Attribute Values

The interfaces described to represent MathML elements include access to a number of attributes (in the sense of XML) belonging to those elements. The intent of these methods in the core MathML interfaces (the 'get'/ 'set' pairs) is only to access *explicitly specified* attributes of the elements, and specifically *not* to access implicit values which may be application-specific. Calls to these interfaces to get attributes that have not been explicitly specified should return nothing (an empty `DOMString`).

It seems important to belabor this distinction in light of the nature of the MathML elements and their attributes; all of the attributes defined for MathML presentation elements are declared in the DTD with a default value of `#IMPLIED`, for instance. This is particularly relevant for the interface of the `mo` element, where the `form` attribute may be inferred from context if not given explicitly, but other attributes are normally collected from an operator dictionary available to a renderer. The variety of applications which may need to implement the MathML DOM may sometimes be concerned with validation, computation or other aspects of the document to the exclusion of rendering or editing; such applications do not need to resolve many `#IMPLIED` attributes, and thus there is no access to such resolution implied in this version of the MathML DOM.

On the other hand, methods for obtaining the current cascaded and computed values of certain style attributes are considered desirable due to the need to make frequent calls to discover style information and the current script level and display style. Mathematics is characterized by recursive nesting of objects, frequently with implications for the calculation of style parameters such as font size. As anyone who's implemented math rendering knows, there's a constant need for this information, and it must be obtained very quickly. Consequently, it might be wise to provide an optional module in the MathML DOM which would allow style values or implied attributes (e.g., operator dictionary values) known to the processing application to be 'attached' to a DOM instance and subsequently queried.

However, we feel that introducing methods now for dealing with these issues would be premature. CSS and XSL support for mathematics is still evolving, and the mechanisms for handling style issues in MathML documents may well evolve with them. Additionally, these issues also apply to the core XML DOM. Thus far (XML DOM level 2), issues such as privacy with regard to user-side style sheets have resulted in no core DOM methods being defined for obtaining the cascaded, computed or actual style values for a specific element, with DOM access being limited to providing the style declarations which are in effect. If a future iteration of the XML DOM were to expand this access, the methods used there would apply to the MathML DOM as well, and render any specifications we might make now obsolete.

8.1.1.2 Traversal and Range Interfaces

Additionally, it is likely that a need will become obvious for MathML-specific specializations of interfaces belonging to the Traversal and Range Modules of XML DOM Level 2. The order of traversal of bound variables, conditions, and declarations - or whether they should be omitted from a given traversal altogether - offers an example of a potential utility for such specializations. Again, however, we feel that it would be premature to specify any such interfaces at this time. Implementation experience will be necessary in order to discover the appropriate interfaces which should be specified.

Appendix A

Parsing MathML

MathML documents should be validated using the XML DTD for MathML, which is also shown below in Section A.1.

Normally. however, a MathML expression does not constitute an entire XML document. MathML is designed to be used as the mathematics fragment of larger markup languages. In particular it is designed to be used as a *module* in documents marked up with the XHTML family of markup languages, as defined in [Modularization]. As a convenience, a version of the XHTML DTD, extended with this MathML module, is also provided as a concrete example. This version includes all the necessary declarations included into one file. (In contrast to the standalone version of the MathML DTD which references several files for entity declarations etc.

In some circumstances, when embedding MathML in documents it is necessary, or convenient, to use the mechanisms described in Chapter 7 which provide a namespace prefix on MathML element names. The DTD below is designed to allow this usage. If the parameter entity `MATHML.prefixed` is declared to be INCLUDE, using a declaration such as

```
<!ENTITY % MATHML.prefixed "INCLUDE" >
```

either in the local subset of the DOCTYPE declaration, or in the DTD file that is including the MathML DTD, then all MathML elements should be used with a prefix, for example `<m:mrow>`, `<m:apply>`, etc. The prefix defaults to `m:` but another prefix may be declared by declaring in addition the parameter entity `MathML.prefix`. For example,

```
<!ENTITY % MATHML.prefix   "math" >
```

would set the prefix for the MathML namespace to `math:`.

Note that while the [Namespaces] Recommendation provides mechanisms to change the prefix at arbitrary points in the document, this flexibility is *not provided* in this DTD (and is probably not possible to specify in any DTD).

If a MathML fragment is parsed without a DTD, in other words as a well-formed XML fragment, it is the responsibility of the processing application to treat the whitespace whitespace characters occurring outside of token elements as not significant.

An SGML parser, such as `nsgmls`, can be used to validate MathML. In this case an SGML declaration defining the constraints of XML applicable to an SGML parser must be used. See the note on SGML and XML.

The entity declarations for characters are referenced at the end of the DTD. These are linked to the character tables in Chapter 6 for each entity set.

Lists of the combined MathML set of character names, ordered by name or by Unicode value are also available.

In order to accommodate XML Namespace prefixes, the DTD does not directly refer to an element name such as `mrow` but instead always refers to the name via a parameter entity such as `%mrow.qname;`. The definitions of these parameter entities are in the file but are not shown here. They are simply declarations such as the following, one for each MathML element.

```
<!ENTITY % mrow.qname           "%MATHML.pfx;mrow" >
```

A.1 The MathML DTD

Here we give the main body of the DTD. The full DTD, as well as the XHTML-Math DTD, is available as a zip archive.

```
<!-- MathML 2.0 DTD  ....................................................... -->
<!-- file: mathml2.dtd
-->

<!-- MathML 2.0 DTD

     This is the Mathematical Markup Language (MathML) 2.0, an XML
     application for describing mathematical notation and capturing
     both its structure and content.

     Copyright 1998-2000 World Wide Web Consortium
        (Massachusetts Institute of Technology, Institut National de
         Recherche en Informatique et en Automatique, Keio University).
         All Rights Reserved.

     Permission to use, copy, modify and distribute the XHTML 1.1 DTD and
     its accompanying documentation for any purpose and without fee is
     hereby granted in perpetuity, provided that the above copyright notice
     and this paragraph appear in all copies.  The copyright holders make
     no representation about the suitability of the DTD for any purpose.

     It is provided "as is" without expressed or implied warranty.

        Revision:   $Id: parsing.xml,v 1.22 2000/03/28 09:53:02 davidc Exp $

     This entity may be identified by the PUBLIC and SYSTEM identifiers:

       PUBLIC "-//W3C//DTD MathML 2.0//EN"
       SYSTEM "mathml2.dtd"

     Revisions: editor and revision history at EOF
-->


<!-- MathML Qualified Names module ............................. -->
  <!ENTITY % mathml-qname.module "INCLUDE" >
<![%mathml-qname.module;[
  <!ENTITY % mathml-qname.mod
     PUBLIC "-//W3C//ENTITIES MathML 2.0 Qualified Names 1.0//EN"
            "mathml2-qname-1.mod" >
%mathml-qname.mod;]]>

<!-- if %NS.prefixed; is INCLUDE, include all NS attributes,
     otherwise just those associated with MathML
```

```
-->
<![%NS.prefixed;[
<!ENTITY % MATHML.NamespaceDecl.attrib
     %NamespaceDecl.attrib; >
]]>
  <!ENTITY % MATHML.NamespaceDecl.attrib
     "%MATHML.xmlns.attrib;"
>

<!-- Attributes shared by all elements  ......................... -->

  <!ENTITY % MATHML.Common.attrib
     "%MATHML.NamespaceDecl.attrib;
      xlink:href   CDATA                   #IMPLIED
      class        CDATA                   #IMPLIED
      style        CDATA                   #IMPLIED
      id           ID                      #IMPLIED
      xref         IDREF                   #IMPLIED
      other        CDATA                   #IMPLIED"
>

<!-- Presentation element set  ................................. -->

<!-- Attribute definitions -->

  <!ENTITY % att-fontsize
     "fontsize     CDATA                   #IMPLIED" >
  <!ENTITY % att-fontweight
     "fontweight   ( normal | bold )       #IMPLIED" >
  <!ENTITY % att-fontstyle
     "fontstyle    ( normal | italic )     #IMPLIED" >
  <!ENTITY % att-fontfamily
     "fontfamily   CDATA                   #IMPLIED" >
  <!ENTITY % att-color
     "color        CDATA                   #IMPLIED" >

  <!ENTITY % att-fontinfo
     "%att-fontsize;
      %att-fontweight;
      %att-fontstyle;
      %att-fontfamily;
      %att-color;"
>

  <!ENTITY % att-form
     "form         ( prefix | infix | postfix )  #IMPLIED" >
  <!ENTITY % att-fence
```

```
    "fence        ( true | false )         #IMPLIED" >
  <!ENTITY % att-separator
    "separator    ( true | false )         #IMPLIED" >
  <!ENTITY % att-lspace
    "lspace       CDATA                    #IMPLIED" >
  <!ENTITY % att-rspace
    "rspace       CDATA                    #IMPLIED" >
  <!ENTITY % att-stretchy
    "stretchy     ( true | false )         #IMPLIED" >
  <!ENTITY % att-symmetric
    "symmetric    ( true | false )         #IMPLIED" >
  <!ENTITY % att-maxsize
    "maxsize      CDATA                    #IMPLIED" >
  <!ENTITY % att-minsize
    "minsize      CDATA                    #IMPLIED" >
  <!ENTITY % att-largeop
    "largeop      ( true | false)          #IMPLIED" >
  <!ENTITY % att-movablelimits
    "movablelimits ( true | false )        #IMPLIED" >
  <!ENTITY % att-accent
    "accent       ( true | false )         #IMPLIED" >

  <!ENTITY % att-opinfo
    "%att-form;
     %att-fence;
     %att-separator;
     %att-lspace;
     %att-rspace;
     %att-stretchy;
     %att-symmetric;
     %att-maxsize;
     %att-minsize;
     %att-largeop;
     %att-movablelimits;
     %att-accent;"
>
  <!ENTITY % att-width
    "width        CDATA                    #IMPLIED" >
  <!ENTITY % att-height
    "height       CDATA                    #IMPLIED" >
  <!ENTITY % att-depth
    "depth        CDATA                    #IMPLIED" >
  <!ENTITY % att-linebreak
    "linebreak    CDATA                    #IMPLIED" >
  <!ENTITY % att-sizeinfo
    "%att-width;
     %att-height;
```

```
      %att-depth;"
>
  <!ENTITY % att-lquote
     "lquote       CDATA                    #IMPLIED" >
  <!ENTITY % att-rquote
     "rquote       CDATA                    #IMPLIED" >
  <!ENTITY % att-linethickness
     "linethickness CDATA                   #IMPLIED" >
  <!ENTITY % att-scriptlevel
     "scriptlevel  CDATA                    #IMPLIED" >
  <!ENTITY % att-displaystyle
     "displaystyle ( true | false )         #IMPLIED" >
  <!ENTITY % att-scriptsizemultiplier
     "scriptsizemultiplier CDATA            #IMPLIED" >
  <!ENTITY % att-scriptminsize
     "scriptminsize CDATA                   #IMPLIED" >
  <!ENTITY % att-background
     "background   CDATA                    #IMPLIED" >
  <!ENTITY % att-open
     "open         CDATA                    #IMPLIED" >
  <!ENTITY % att-close
     "close        CDATA                    #IMPLIED" >
  <!ENTITY % att-separators
     "separators   CDATA                    #IMPLIED" >
  <!ENTITY % att-subscriptshift
     "subscriptshift CDATA                  #IMPLIED" >
  <!ENTITY % att-superscriptshift
     "superscriptshift CDATA                #IMPLIED" >
  <!ENTITY % att-accentunder
     "accentunder  ( true | false )         #IMPLIED" >
  <!ENTITY % att-align
     "align        CDATA                    #IMPLIED" >
  <!ENTITY % att-rowalign
     "rowalign     CDATA                    #IMPLIED" >
  <!ENTITY % att-columnalign
     "columnalign  CDATA                    #IMPLIED" >
  <!ENTITY % att-columnwidth
     "columnwidth  CDATA                    #IMPLIED" >
  <!ENTITY % att-groupalign
     "groupalign   CDATA                    #IMPLIED" >
  <!ENTITY % att-alignmentscope
     "alignmentscope CDATA                  #IMPLIED" >
  <!ENTITY % att-rowspacing
     "rowspacing   CDATA                    #IMPLIED" >
  <!ENTITY % att-columnspacing
     "columnspacing CDATA                   #IMPLIED" >
  <!ENTITY % att-rowlines
```

```
    "rowlines     CDATA                   #IMPLIED" >
 <!ENTITY % att-columnlines
    "columnlines  CDATA                   #IMPLIED" >
 <!ENTITY % att-frame
    "frame       ( none | solid | dashed ) #IMPLIED" >
 <!ENTITY % att-framespacing
    "framespacing CDATA                   #IMPLIED" >
 <!ENTITY % att-equalrows
    "equalrows    CDATA                   #IMPLIED" >
 <!ENTITY % att-equalcolumns
    "equalcolumns CDATA                   #IMPLIED" >

 <!ENTITY % att-tableinfo
    "%att-align;
     %att-rowalign;
     %att-columnalign;
     %att-columnwidth;
     %att-groupalign;
     %att-alignmentscope;
     %att-rowspacing;
     %att-columnspacing;
     %att-rowlines;
     %att-columnlines;
     %att-frame;
     %att-framespacing;
     %att-equalrows;
     %att-equalcolumns;
     %att-displaystyle;"
>

 <!ENTITY % att-rowspan
    "rowspan      CDATA                   #IMPLIED" >
 <!ENTITY % att-columnspan
    "columnspan   CDATA                   #IMPLIED" >
 <!ENTITY % att-edge
    "edge         ( left | right )        #IMPLIED" >
 <!ENTITY % att-actiontype
    "actiontype   CDATA                   #IMPLIED" >
 <!ENTITY % att-selection
    "selection    CDATA                   #IMPLIED" >

 <!ENTITY % att-name
    "name         CDATA                   #IMPLIED" >
 <!ENTITY % att-alt
    "alt          CDATA                   #IMPLIED" >
 <!ENTITY % att-index
    "index        CDATA                   #IMPLIED" >
```

```
  <!ENTITY % att-bevelled
     "bevelled      CDATA                   #IMPLIED" >

<!-- Presentation schemata with content -->

  <!ENTITY % ptoken
     "%mi.qname; | %mn.qname; | %mo.qname;
      | %mtext.qname; | %ms.qname;" >

  <!ATTLIST %mi.qname;
      %MATHML.Common.attrib;
      %att-fontinfo;
>

  <!ATTLIST %mn.qname;
      %MATHML.Common.attrib;
      %att-fontinfo;
>

  <!ATTLIST %mo.qname;
      %MATHML.Common.attrib;
      %att-fontinfo;
      %att-opinfo;
>

  <!ATTLIST %mtext.qname;
      %MATHML.Common.attrib;
      %att-fontinfo;
>

  <!ATTLIST %ms.qname;
      %MATHML.Common.attrib;
      %att-fontinfo;
      %att-lquote;
      %att-rquote;
>

<!-- Empty presentation schemata -->

  <!ENTITY % petoken
     "%mspace.qname;" >
  <!ELEMENT %mspace.qname;  EMPTY >

  <!ATTLIST %mspace.qname;
      %att-sizeinfo;
      %att-linebreak;
```

```
      %MATHML.Common.attrib;
>

<!-- Presentation: general layout schemata -->

  <!ENTITY % pgenschema
     "%mrow.qname; | %mfrac.qname; | %msqrt.qname; | %mroot.qname;
      | %menclose.qname; | %mstyle.qname; | %merror.qname;
      | %mpadded.qname; | %mphantom.qname; | %mfenced.qname;" >

  <!ATTLIST %mrow.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %mfrac.qname;
      %MATHML.Common.attrib;
      %att-bevelled;
      %att-linethickness;
>

  <!ATTLIST %msqrt.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %menclose.qname;
      %MATHML.Common.attrib;
      notation CDATA 'longdiv' >

  <!ATTLIST %mroot.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %mstyle.qname;
      %MATHML.Common.attrib;
      %att-fontinfo;
      %att-opinfo;
      %att-lquote;
      %att-rquote;
      %att-linethickness;
      %att-scriptlevel;
      %att-scriptsizemultiplier;
      %att-scriptminsize;
      %att-background;
      %att-open;
      %att-close;
      %att-separators;
      %att-subscriptshift;
```

```
      %att-superscriptshift;
      %att-accentunder;
      %att-tableinfo;
      %att-rowspan;
      %att-columnspan;
      %att-edge;
      %att-actiontype;
      %att-selection;
>

  <!ATTLIST %merror.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %mpadded.qname;
      %MATHML.Common.attrib;
      %att-sizeinfo;
      %att-lspace;
>

  <!ATTLIST %mphantom.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %mfenced.qname;
      %MATHML.Common.attrib;
      %att-open;
      %att-close;
      %att-separators;
>

<!-- Presentation layout schemata: scripts and limits -->

  <!ENTITY % pscrschema
     "%msub.qname; | %msup.qname; | %msubsup.qname; | %munder.qname;
      | %mover.qname; | %munderover.qname; | %mmultiscripts.qname;" >

  <!ATTLIST %msub.qname;
      %MATHML.Common.attrib;
      %att-subscriptshift;
>

  <!ATTLIST %msup.qname;
      %MATHML.Common.attrib;
      %att-superscriptshift;
>
```

```
  <!ATTLIST %msubsup.qname;
      %MATHML.Common.attrib;
      %att-subscriptshift;
      %att-superscriptshift;
>

  <!ATTLIST %munder.qname;
      %MATHML.Common.attrib;
      %att-accentunder;
>

  <!ATTLIST %mover.qname;
      %MATHML.Common.attrib;
      %att-accent;
>

  <!ATTLIST %munderover.qname;
      %MATHML.Common.attrib;
      %att-accent;
      %att-accentunder;
>

  <!ATTLIST %mmultiscripts.qname;
      %MATHML.Common.attrib;
      %att-subscriptshift;
      %att-superscriptshift;
>

<!-- Presentation layout schemata: empty elements for scripts -->

  <!ENTITY % pscreschema
     "%mprescripts.qname; | %none.qname;" >

  <!ELEMENT %mprescripts.qname;  EMPTY >
  <!ATTLIST %mprescripts.qname;
      %MATHML.xmlns.attrib; >

  <!ELEMENT %none.qname;  EMPTY >
  <!ATTLIST %none.qname;
      %MATHML.xmlns.attrib; >

<!-- Presentation layout schemata: tables -->

  <!ENTITY % ptabschema
     "%mtable.qname; | %mtr.qname; | %mlabeledtr.qname; | %mtd.qname;" >

  <!ATTLIST %mtable.qname;
```

```
      %MATHML.Common.attrib;
      %att-tableinfo;
>

  <!ATTLIST %mtr.qname;
      %MATHML.Common.attrib;
      %att-rowalign;
      %att-columnalign;
      %att-groupalign;
>

  <!ATTLIST %mlabeledtr.qname;
      %MATHML.Common.attrib;
      %att-rowalign;
      %att-columnalign;
      %att-groupalign;
>

  <!ATTLIST %mtd.qname;
      %MATHML.Common.attrib;
      %att-rowalign;
      %att-columnalign;
      %att-groupalign;
      %att-rowspan;
      %att-columnspan;
>

  <!ENTITY % plschema
     "%pgenschema; | %pscrschema; | %ptabschema;" >

<!-- Empty presentation layout schemata -->

  <!ENTITY % peschema
     "%maligngroup.qname; | %malignmark.qname;" >

  <!ELEMENT %malignmark.qname;  EMPTY >

  <!ATTLIST %malignmark.qname;
      %att-edge; >

  <!ELEMENT %maligngroup.qname;  EMPTY >
  <!ATTLIST %maligngroup.qname;
      %MATHML.Common.attrib;
      %att-groupalign;
>

  <!ELEMENT %mchar.qname;  EMPTY >
```

```
  <!ATTLIST %mchar.qname;
      %att-name; >

  <!ELEMENT %mglyph.qname;  EMPTY >
  <!ATTLIST %mglyph.qname;
      %att-alt;
      %att-fontfamily;
      %att-index; >

<!-- Presentation action schemata -->

  <!ENTITY % pactions
     "%maction.qname;" >
  <!ATTLIST %maction.qname;
      %MATHML.Common.attrib;
      %att-actiontype;
      %att-selection;
>

<!-- The following entity for substitution into
     content constructs excludes elements that
     are not valid as expressions.
-->

  <!ENTITY % PresInCont
     "%ptoken; | %petoken; |
      %plschema; | %peschema; | %pactions;" >

<!-- Presentation entity: all presentation constructs -->

  <!ENTITY % Presentation
     "%ptoken; | %petoken; | %pscreschema; |
      %plschema; | %peschema; | %pactions;">

<!-- Content element set  ...................................... -->

<!-- Attribute definitions -->

  <!ENTITY % att-base
     "base         CDATA                    '10'" >
  <!ENTITY % att-closure
     "closure      CDATA                    'closed'" >
  <!ENTITY % att-definition
     "definitionURL CDATA                   ''" >
  <!ENTITY % att-encoding
     "encoding     CDATA                    ''" >
  <!ENTITY % att-nargs
```

```
    "nargs        CDATA                    '1'" >
  <!ENTITY % att-occurrence
    "occurrence   CDATA                    'function-model'" >
  <!ENTITY % att-order
    "order        CDATA                    'numeric'" >
  <!ENTITY % att-scope
    "scope        CDATA                    'local'" >
  <!ENTITY % att-type
    "type         CDATA                    #IMPLIED" >

<!-- Content elements: leaf nodes -->

  <!ENTITY % ctoken
    "%csymbol.qname; | %ci.qname; | %cn.qname;" >

  <!ATTLIST %ci.qname;
     %MATHML.Common.attrib;
     %att-type;
     %att-definition;
     %att-encoding;
>

  <!ATTLIST %csymbol.qname;
     %MATHML.Common.attrib;
     %att-encoding;
     %att-type;
     %att-definition;
>

  <!ATTLIST %cn.qname;
     %MATHML.Common.attrib;
     %att-type;
     %att-base;
     %att-definition;
     %att-encoding;
>

<!-- Content elements: specials -->

  <!ENTITY % cspecial
    "%apply.qname; | %reln.qname; |
     %lambda.qname;" >

  <!ATTLIST %apply.qname;
     %MATHML.Common.attrib;
>
```

```
  <!ATTLIST %reln.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %lambda.qname;
      %MATHML.Common.attrib;
>

<!-- Content elements: others -->

  <!ENTITY % cother
     "%condition.qname; | %declare.qname; | %sep.qname;" >

  <!ATTLIST %condition.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %declare.qname;
      %MATHML.Common.attrib;
      %att-type;
      %att-scope;
      %att-nargs;
      %att-occurrence;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %sep.qname;  EMPTY >
  <!ATTLIST %sep.qname;
      %MATHML.xmlns.attrib; >

<!-- Content elements: semantic mapping -->

  <!ENTITY % csemantics
     "%semantics.qname; | %annotation.qname; |
      %annotation-xml.qname;" >

  <!ATTLIST %semantics.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ATTLIST %annotation.qname;
      %MATHML.Common.attrib;
      %att-encoding;
>
```

```
  <!ATTLIST %annotation-xml.qname;
      %MATHML.Common.attrib;
      %att-encoding;
>

<!-- Content elements: constructors -->

  <!ENTITY % cconstructor
     "%interval.qname; | %list.qname; | %matrix.qname;
      | %matrixrow.qname; | %set.qname; | %vector.qname;" >

  <!ATTLIST %interval.qname;
      %MATHML.Common.attrib;
      %att-closure;
>

  <!ATTLIST %set.qname;
      %MATHML.Common.attrib;
      %att-type;
>

  <!ATTLIST %list.qname;
      %MATHML.Common.attrib;
      %att-order;
>

  <!ATTLIST %vector.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %matrix.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %matrixrow.qname;
      %MATHML.Common.attrib;
>

<!-- Content elements: symbols -->

  <!ENTITY % c0ary
    "%integers.qname; |
     %reals.qname; |
     %rationals.qname; |
     %naturalnumbers.qname; |
     %complexes.qname; |
```

```
     %primes.qname; |
     %exponentiale.qname; |
     %imaginaryi.qname; |
     %notanumber.qname; |
     %true.qname; |
     %false.qname; |
     %emptyset.qname; |
     %pi.qname; |
     %eulergamma.qname; |
     %infinity.qname;" >

<!-- Content elements: operators -->

  <!ENTITY % cfuncop1ary
     "%inverse.qname; | %ident.qname;" >

  <!ELEMENT %inverse.qname;  EMPTY >
  <!ATTLIST %inverse.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % cfuncopnary
     "%fn.qname; | %compose.qname;" >

  <!ATTLIST %fn.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %ident.qname;  EMPTY >
  <!ATTLIST %ident.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %compose.qname;  EMPTY >
  <!ATTLIST %compose.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % carithop1ary
```

```
    "%abs.qname; | %conjugate.qname; | %exp.qname; | %factorial.qname; |
     %arg.qname; | %real.qname; | %imaginary.qname;" >

  <!ELEMENT %exp.qname;  EMPTY >
  <!ATTLIST %exp.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %abs.qname;  EMPTY >
  <!ATTLIST %abs.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %arg.qname;  EMPTY >
  <!ATTLIST %arg.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %real.qname;  EMPTY >
  <!ATTLIST %real.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %imaginary.qname;  EMPTY >
  <!ATTLIST %imaginary.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %conjugate.qname;  EMPTY >
  <!ATTLIST %conjugate.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %factorial.qname;  EMPTY >
  <!ATTLIST %factorial.qname;
```

```
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % carithop1or2ary
     "%minus.qname;" >

  <!ELEMENT %minus.qname;  EMPTY >
  <!ATTLIST %minus.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % carithop2ary
     "%quotient.qname; | %divide.qname; | %power.qname; | %rem.qname;" >

  <!ELEMENT %quotient.qname;  EMPTY >
  <!ATTLIST %quotient.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %divide.qname;  EMPTY >
  <!ATTLIST %divide.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %power.qname;  EMPTY >
  <!ATTLIST %power.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %rem.qname;  EMPTY >
  <!ATTLIST %rem.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % carithopnary
```

```
     "%plus.qname; | %times.qname; | %max.qname;
      | %min.qname; | %gcd.qname; | %lcm.qname;" >

  <!ELEMENT %plus.qname;  EMPTY >
  <!ATTLIST %plus.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %max.qname;  EMPTY >
  <!ATTLIST %max.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %min.qname;  EMPTY >
  <!ATTLIST %min.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %times.qname;  EMPTY >
  <!ATTLIST %times.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %gcd.qname;  EMPTY >
  <!ATTLIST %gcd.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %lcm.qname;  EMPTY >
  <!ATTLIST %lcm.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % carithoproot
     "%root.qname;" >
```

```
  <!ELEMENT %root.qname;  EMPTY >
  <!ATTLIST %root.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % clogicopquant
     "%exists.qname; | %forall.qname;" >

  <!ELEMENT %exists.qname;  EMPTY >
  <!ATTLIST %exists.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %forall.qname;  EMPTY >
  <!ATTLIST %forall.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % clogicopnary
     "%and.qname; | %or.qname; | %xor.qname;" >

  <!ELEMENT %and.qname;  EMPTY >
  <!ATTLIST %and.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %or.qname;  EMPTY >
  <!ATTLIST %or.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %xor.qname;  EMPTY >
  <!ATTLIST %xor.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
```

```
>

  <!ENTITY % clogicop1ary
     "%not.qname;" >

  <!ELEMENT %not.qname;  EMPTY >
  <!ATTLIST %not.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % clogicop2ary
     "%implies.qname;" >

  <!ELEMENT %implies.qname;  EMPTY >
  <!ATTLIST %implies.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % ccalcop
     "%log.qname; | %int.qname; | %diff.qname; | %partialdiff.qname; |
      %divergence.qname; | %grad.qname; | %curl.qname; | %laplacian.qname;" >

  <!ELEMENT %divergence.qname;  EMPTY >
  <!ATTLIST %divergence.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %grad.qname;  EMPTY >
  <!ATTLIST %grad.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %curl.qname;  EMPTY >
  <!ATTLIST %curl.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>
```

```
  <!ELEMENT %laplacian.qname;  EMPTY >
  <!ATTLIST %laplacian.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %log.qname;  EMPTY >
  <!ATTLIST %log.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %int.qname;  EMPTY >
  <!ATTLIST %int.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %diff.qname;  EMPTY >
  <!ATTLIST %diff.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %partialdiff.qname;  EMPTY >
  <!ATTLIST %partialdiff.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % ccalcop1ary
     "%ln.qname;" >

  <!ELEMENT %ln.qname;  EMPTY >
  <!ATTLIST %ln.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % csetop1ary
     "%card.qname;" >
```

```
  <!ELEMENT %card.qname;  EMPTY >
  <!ATTLIST %card.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % csetop2ary
     "%setdiff.qname;" >

  <!ELEMENT %setdiff.qname;  EMPTY >
  <!ATTLIST %setdiff.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % csetopnary
     "%union.qname; | %intersect.qname;" >

  <!ELEMENT %union.qname;  EMPTY >
  <!ATTLIST %union.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %intersect.qname;  EMPTY >
  <!ATTLIST %intersect.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % cseqop
     "%sum.qname; | %product.qname; | %limit.qname;" >

  <!ELEMENT %sum.qname;  EMPTY >
  <!ATTLIST %sum.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %product.qname;  EMPTY >
  <!ATTLIST %product.qname;
```

```
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %limit.qname;  EMPTY >
  <!ATTLIST %limit.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % ctrigop
     "%sin.qname; | %cos.qname; | %tan.qname;
      | %sec.qname; | %csc.qname; | %cot.qname;
      | %sinh.qname; | %cosh.qname; | %tanh.qname;
      | %sech.qname; | %csch.qname; | %coth.qname;
      | %arcsin.qname; | %arccos.qname; | %arctan.qname;" >

  <!ELEMENT %sin.qname;  EMPTY >
  <!ATTLIST %sin.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %cos.qname;  EMPTY >
  <!ATTLIST %cos.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %tan.qname;  EMPTY >
  <!ATTLIST %tan.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %sec.qname;  EMPTY >
  <!ATTLIST %sec.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>
```

```
  <!ELEMENT %csc.qname;  EMPTY >
  <!ATTLIST %csc.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %cot.qname;  EMPTY >
  <!ATTLIST %cot.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %sinh.qname;  EMPTY >
  <!ATTLIST %sinh.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %cosh.qname;  EMPTY >
  <!ATTLIST %cosh.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %tanh.qname;  EMPTY >
  <!ATTLIST %tanh.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %sech.qname;  EMPTY >
  <!ATTLIST %sech.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %csch.qname;  EMPTY >
  <!ATTLIST %csch.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
```

```
>

  <!ELEMENT %coth.qname;  EMPTY >
  <!ATTLIST %coth.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %arcsin.qname;  EMPTY >
  <!ATTLIST %arcsin.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %arccos.qname;  EMPTY >
  <!ATTLIST %arccos.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %arctan.qname;  EMPTY >
  <!ATTLIST %arctan.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % cstatopnary
     "%mean.qname; | %sdev.qname; |
      %variance.qname; | %median.qname; |
      %mode.qname;" >

  <!ELEMENT %mean.qname;  EMPTY >
  <!ATTLIST %mean.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %sdev.qname;  EMPTY >
  <!ATTLIST %sdev.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
```

```
>

  <!ELEMENT %variance.qname;  EMPTY >
  <!ATTLIST %variance.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %median.qname;  EMPTY >
  <!ATTLIST %median.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %mode.qname;  EMPTY >
  <!ATTLIST %mode.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % cstatopmoment
     "%moment.qname;" >

  <!ELEMENT %moment.qname;  EMPTY >
  <!ATTLIST %moment.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % calgop1ary
     "%determinant.qname; |
      %transpose.qname;" >

  <!ELEMENT %determinant.qname;  EMPTY >
  <!ATTLIST %determinant.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %transpose.qname;  EMPTY >
  <!ATTLIST %transpose.qname;
      %MATHML.Common.attrib;
```

```
      %att-definition;
      %att-encoding;
>

  <!ENTITY % clalgop2ary
     "%vectorproduct.qname;
      | %scalarproduct.qname;
      | %outerproduct.qname;" >

  <!ELEMENT %vectorproduct.qname;  EMPTY >
  <!ATTLIST %vectorproduct.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %scalarproduct.qname;  EMPTY >
  <!ATTLIST %scalarproduct.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %outerproduct.qname;  EMPTY >
  <!ATTLIST %outerproduct.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % clalgopnary
     "%selector.qname;" >

  <!ELEMENT %selector.qname;  EMPTY >
  <!ATTLIST %selector.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

<!-- Content elements: relations -->

  <!ENTITY % cgenrel2ary
     "%neq.qname;" >

  <!ELEMENT %neq.qname;  EMPTY >
  <!ATTLIST %neq.qname;
```

```
    %MATHML.Common.attrib;
    %att-definition;
    %att-encoding;
>

  <!ENTITY % cgenrelnary
     "%eq.qname; | %leq.qname; | %lt.qname; | %geq.qname;
      | %gt.qname;| %equivalent.qname; | %approx.qname;" >

  <!ELEMENT %eq.qname;  EMPTY >
  <!ATTLIST %eq.qname;
    %MATHML.Common.attrib;
    %att-definition;
    %att-encoding;
>

  <!ELEMENT %equivalent.qname;  EMPTY >
  <!ATTLIST %equivalent.qname;
    %MATHML.Common.attrib;
    %att-definition;
    %att-encoding;
>

  <!ELEMENT %approx.qname;  EMPTY >
  <!ATTLIST %approx.qname;
    %MATHML.Common.attrib;
    %att-definition;
    %att-encoding;
>

  <!ELEMENT %gt.qname;  EMPTY >
  <!ATTLIST %gt.qname;
    %MATHML.Common.attrib;
    %att-definition;
    %att-encoding;
>

  <!ELEMENT %lt.qname;  EMPTY >
  <!ATTLIST %lt.qname;
    %MATHML.Common.attrib;
    %att-definition;
    %att-encoding;
>

  <!ELEMENT %geq.qname;  EMPTY >
  <!ATTLIST %geq.qname;
    %MATHML.Common.attrib;
```

```
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %leq.qname;  EMPTY >
  <!ATTLIST %leq.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % csetrel2ary
     "%in.qname; | %notin.qname; | %notsubset.qname; | %notprsubset.qname;" >

  <!ELEMENT %in.qname;  EMPTY >
  <!ATTLIST %in.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %notin.qname;  EMPTY >
  <!ATTLIST %notin.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %notsubset.qname;  EMPTY >
  <!ATTLIST %notsubset.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %notprsubset.qname;  EMPTY >
  <!ATTLIST %notprsubset.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % csetrelnary
     "%subset.qname; | %prsubset.qname;" >

  <!ELEMENT %subset.qname;  EMPTY >
  <!ATTLIST %subset.qname;
```

```
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ELEMENT %prsubset.qname;  EMPTY >
  <!ATTLIST %prsubset.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
>

  <!ENTITY % cseqrel2ary
     "%tendsto.qname;" >

  <!ELEMENT %tendsto.qname;  EMPTY >
  <!ATTLIST %tendsto.qname;
      %MATHML.Common.attrib;
      %att-definition;
      %att-encoding;
      %att-type;
>

<!-- Content elements: quantifiers -->

  <!ENTITY % cquantifier
     "%lowlimit.qname; | %uplimit.qname; | %bvar.qname;
      | %degree.qname; | %logbase.qname;" >

  <!ATTLIST %lowlimit.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %uplimit.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %bvar.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %degree.qname;
      %MATHML.Common.attrib;
>

  <!ATTLIST %logbase.qname;
      %MATHML.Common.attrib;
```

```
>

<!-- Operator groups -->

  <!ENTITY % cop1ary
     "%cfuncop1ary; | %carithop1ary; | %clogicop1ary; |
      %ccalcop1ary; | %ctrigop; | %clalgop1ary; |
      %csetop1ary;" >

  <!ENTITY % cop2ary
     "%carithop2ary; | %clogicop2ary;| %clalgop2ary; | %csetop2ary;" >

  <!ENTITY % copnary
     "%cfuncopnary; | %carithopnary; | %clogicopnary; |
      %csetopnary; | %cstatopnary; | %clalgopnary;" >

  <!ENTITY % copmisc
     "%carithoproot; | %carithop1or2ary; | %ccalcop; |
      %cseqop; | %cstatopmoment; | %clogicopquant;" >

<!-- Relation groups -->

  <!ENTITY % crel2ary
     "%cgenrel2ary; | %csetrel2ary; | %cseqrel2ary;" >

  <!ENTITY % crelnary
     "%cgenrelnary; | %csetrelnary;" >

<!-- Content constructs: all -->

  <!ENTITY % Content
     "%ctoken; | %cspecial; | %cother; | %csemantics; | %c0ary;
      | %cconstructor; | %cquantifier; | %cop1ary; | %cop2ary;
      | %copnary; |%copmisc; | %crel2ary; | %crelnary;" >

<!-- Content constructs for substitution in presentation structures -->

  <!ENTITY % ContInPres
     "%ci.qname; |%csymbol.qname;| %cn.qname;|
      %apply.qname; | %fn.qname; |
      %lambda.qname; | %reln.qname; |
      %interval.qname; | %list.qname; |
      %matrix.qname; | %matrixrow.qname; |
      %set.qname; | %vector.qname; |
      %semantics.qname; |%declare.qname;" >

<!-- ...................................................... -->
```

```
<!-- Recursive definition for content of expressions. Include
     presentation constructs at lowest level so presentation
     layout schemata hold presentation or content elements.
     Include content constructs at lowest level so content
     elements hold PCDATA or presentation elements at leaf
     level (for permitted substitutable elements in context)
-->

  <!ENTITY % ContentExpression
     "(%Content; | %PresInCont;)*" >
  <!ENTITY % PresExpression
     "(%Presentation; | %ContInPres;)*" >
  <!ENTITY % MathExpression
     "(%PresInCont; | %ContInPres;)*" >

<!-- PCDATA or MathML character elements -->
  <!ENTITY % MathMLCharacters
     "#PCDATA | %mchar.qname; | %mglyph.qname; " >

<!-- Content elements: tokens                       -->
<!-- (may contain embedded presentation constructs) -->

  <!ELEMENT %ci.qname;                (%MathMLCharacters; | %PresInCont;)* >
  <!ELEMENT %csymbol.qname;           (%MathMLCharacters; | %PresInCont;)* >
  <!ELEMENT %cn.qname;                (%MathMLCharacters; | %sep.qname; | %PresInCont;)* >

<!-- Content elements: special -->

  <!ELEMENT %apply.qname;             (%ContentExpression;) >
  <!ELEMENT %reln.qname;              (%ContentExpression;) >
  <!ELEMENT %lambda.qname;            (%ContentExpression;) >

<!-- Content elements: other -->

  <!ELEMENT %condition.qname;         (%ContentExpression;) >
  <!ELEMENT %declare.qname;           (%ContentExpression;) >

<!-- Content elements: semantics -->

  <!ELEMENT %semantics.qname;         (%ContentExpression;) >
  <!ENTITY % Annotation.content  "( #PCDATA )" >
  <!ELEMENT %annotation.qname;        %Annotation.content; >

  <!ENTITY % Annotation-xml.content "ANY" >
  <!ELEMENT %annotation-xml.qname;    %Annotation-xml.content; >

<!-- Content elements: constructors -->
```

```
  <!ELEMENT %interval.qname;          (%ContentExpression;) >
  <!ELEMENT %set.qname;               (%ContentExpression;) >
  <!ELEMENT %list.qname;              (%ContentExpression;) >
  <!ELEMENT %vector.qname;            (%ContentExpression;) >
  <!ELEMENT %matrix.qname;            (%ContentExpression;) >
  <!ELEMENT %matrixrow.qname;         (%ContentExpression;) >

<!-- Content elements: operator (user-defined) -->

  <!ELEMENT %fn.qname;                (%ContentExpression;) >

<!-- Content elements: quantifiers -->

  <!ELEMENT %lowlimit.qname;          (%ContentExpression;) >
  <!ELEMENT %uplimit.qname;           (%ContentExpression;) >
  <!ELEMENT %bvar.qname;              (%ContentExpression;) >
  <!ELEMENT %degree.qname;            (%ContentExpression;) >
  <!ELEMENT %logbase.qname;           (%ContentExpression;) >

<!-- ............................................................. -->
<!-- Presentation layout schemata contain tokens,
     layout and content schemata.
-->

  <!ELEMENT %mstyle.qname;            (%PresExpression;) >
  <!ELEMENT %merror.qname;            (%PresExpression;) >
  <!ELEMENT %mphantom.qname;          (%PresExpression;) >
  <!ELEMENT %mrow.qname;              (%PresExpression;) >
  <!ELEMENT %mfrac.qname;             (%PresExpression;) >
  <!ELEMENT %msqrt.qname;             (%PresExpression;) >
  <!ELEMENT %menclose.qname;          (%PresExpression;) >
  <!ELEMENT %mroot.qname;             (%PresExpression;) >
  <!ELEMENT %msub.qname;              (%PresExpression;) >
  <!ELEMENT %msup.qname;              (%PresExpression;) >
  <!ELEMENT %msubsup.qname;           (%PresExpression;) >
  <!ELEMENT %mmultiscripts.qname;     (%PresExpression;) >
  <!ELEMENT %munder.qname;            (%PresExpression;) >
  <!ELEMENT %mover.qname;             (%PresExpression;) >
  <!ELEMENT %munderover.qname;        (%PresExpression;) >
  <!ELEMENT %mtable.qname;            (%PresExpression;) >
  <!ELEMENT %mtr.qname;               (%PresExpression;) >
  <!ELEMENT %mlabeledtr.qname;        (%PresExpression;) >
  <!ELEMENT %mtd.qname;               (%PresExpression;) >
  <!ELEMENT %maction.qname;           (%PresExpression;) >
  <!ELEMENT %mfenced.qname;           (%PresExpression;) >
  <!ELEMENT %mpadded.qname;           (%PresExpression;) >
```

```
<!-- Presentation elements contain PCDATA or malignmark constructs. -->

  <!ELEMENT %mi.qname;                 (%MathMLCharacters; |
      %malignmark.qname;)* >
  <!ELEMENT %mn.qname;                 (%MathMLCharacters; |
      %malignmark.qname;)* >
  <!ELEMENT %mo.qname;                 (%MathMLCharacters; |
      %malignmark.qname;)* >
  <!ELEMENT %mtext.qname;              (%MathMLCharacters; |
      %malignmark.qname;)* >
  <!ELEMENT %ms.qname;                 (%MathMLCharacters; |
      %malignmark.qname;)* >

<!-- Browser interface definition  .............................. -->

<!-- Attributes for top-level element "math" -->

  <!ENTITY % att-macros
     "macros       CDATA                    #IMPLIED" >
  <!ENTITY % att-mode
     "mode         CDATA                    #IMPLIED" >

  <!ENTITY % att-topinfo
     "%MATHML.Common.attrib;
      %att-macros;
      %att-mode;" >

<!-- Attributes for browser interface element -->

  <!ENTITY % att-baseline
     "baseline     CDATA                    #IMPLIED" >
  <!ENTITY % att-overflow
     "overflow  ( scroll | elide | truncate | scale ) 'scroll'" >
  <!ENTITY % att-altimg
     "altimg       CDATA                    #IMPLIED" >
  <!ENTITY % att-alttext
     "alttext      CDATA                    #IMPLIED" >

  <!ENTITY % att-browif
     "%att-type;
      %att-name;
      %att-height;
      %att-width;
      %att-baseline;
      %att-overflow;
      %att-altimg;
```

```
      %att-alttext;" >

<!-- ................................................................. -->
<!-- The top-level element "math" contains MathML encoded
     mathematics. The "math" element has the browser info
     attributes iff it is also the browser interface element.
-->

  <!ELEMENT %math.qname;               (%MathExpression;) >

  <!ATTLIST %math.qname;
      %att-topinfo;
      %att-browif; >

<!-- MathML Character Entities ............................................ -->
  <!ENTITY % mathml-charent.module "INCLUDE" >
<![%mathml-charent.module;[
<!-- Entity sets from ISO Technical Report 9573-13 ..... -->

  <!ENTITY % ent-isoamsa
      PUBLIC "-//W3C//ENTITIES Added Math Symbols: Arrow Relations for MathML 2.0//EN"
             "isoamsa.ent" >
%ent-isoamsa;

  <!ENTITY % ent-isoamsb
      PUBLIC "-//W3C//ENTITIES Added Math Symbols: Binary Operators for MathML 2.0//EN"
             "isoamsb.ent" >
%ent-isoamsb;

  <!ENTITY % ent-isoamsc
      PUBLIC "-//W3C//ENTITIES Added Math Symbols: Delimiters for MathML 2.0//EN"
             "isoamsc.ent" >
%ent-isoamsc;

  <!ENTITY % ent-isoamsn
      PUBLIC "-//W3C//ENTITIES Added Math Symbols: Negated Relations for MathML 2.0//EN"
             "isoamsn.ent" >
%ent-isoamsn;

  <!ENTITY % ent-isoamso
      PUBLIC "-//W3C//ENTITIES Added Math Symbols: Ordinary for MathML 2.0//EN"
             "isoamso.ent" >
%ent-isoamso;

  <!ENTITY % ent-isoamsr
      PUBLIC "-//W3C//ENTITIES Added Math Symbols: Relations for MathML 2.0//EN"
             "isoamsr.ent" >
```

```
%ent-isoamsr;

  <!ENTITY % ent-isogrk3
      PUBLIC "-//W3C//ENTITIES Greek Symbols for MathML 2.0//EN"
             "isogrk3.ent" >
%ent-isogrk3;

  <!ENTITY % ent-isomfrk
      PUBLIC "-//W3C//ENTITIES Math Alphabets: Fraktur for MathML 2.0//EN"
             "isomfrk.ent" >
%ent-isomfrk;

  <!ENTITY % ent-isomopf
      PUBLIC "-//W3C//ENTITIES Math Alphabets: Open Face for MathML 2.0//EN"
             "isomopf.ent" >
%ent-isomopf;

  <!ENTITY % ent-isomscr
      PUBLIC "-//W3C//ENTITIES Math Alphabets: Script for MathML 2.0//EN"
             "isomscr.ent" >
%ent-isomscr;

  <!ENTITY % ent-isotech
      PUBLIC "-//W3C//ENTITIES General Technical for MathML 2.0//EN"
             "isotech.ent" >
%ent-isotech;

<!-- Entity sets from informative annex to ISO 8879:1986 (SGML) ....... -->

  <!ENTITY % ent-isobox
      PUBLIC "-//W3C//ENTITIES Box and Line Drawing for MathML 2.0//EN"
             "isobox.ent" >
%ent-isobox;

  <!ENTITY % ent-isocyr1
      PUBLIC "-//W3C//ENTITIES Russian Cyrillic for MathML 2.0//EN"
             "isocyr1.ent" >
%ent-isocyr1;

  <!ENTITY % ent-isocyr2
      PUBLIC "-//W3C//ENTITIES Non-Russian Cyrillic for MathML 2.0//EN"
             "isocyr2.ent" >
%ent-isocyr2;

  <!ENTITY % ent-isodia
      PUBLIC "-//W3C//ENTITIES Diacritical Marks for MathML 2.0//EN"
             "isodia.ent" >
```

```
%ent-isodia;

  <!ENTITY % ent-isolat1
      PUBLIC "-//W3C//ENTITIES Added Latin 1 for MathML 2.0//EN"
             "isolat1.ent" >
%ent-isolat1;

  <!ENTITY % ent-isolat2
      PUBLIC "-//W3C//ENTITIES Added Latin 2 for MathML 2.0//EN"
             "isolat2.ent" >
%ent-isolat2;

  <!ENTITY % ent-isonum
      PUBLIC "-//W3C//ENTITIES Numeric and Special Graphic for MathML 2.0//EN"
             "isonum.ent" >
%ent-isonum;

  <!ENTITY % ent-isopub
      PUBLIC "-//W3C//ENTITIES Publishing for MathML 2.0//EN"
             "isopub.ent" >
%ent-isopub;

<!-- MathML aliases for characters defined above ................. -->

  <!ENTITY % ent-mmlalias
      PUBLIC "-//W3C//ENTITIES Aiases for MathML 2.0//EN"
             "mmlalias.ent" >
<!--%ent-mmlalias;-->

<!-- New characters defined by MathML ............................ -->

  <!ENTITY % ent-mmlextra
      PUBLIC "-//W3C//ENTITIES Extra for MathML 2.0//EN"
             "mmlextra.ent" >
<!--%ent-mmlextra;-->

<!-- end of MathML Character Entity section -->]]>

<!-- Revision History:

       Initial draft (syntax = XML) 1997-05-09
          Stephen Buswell
       Revised 1997-05-14
          Robert Miner
       Revised 1997-06-29 and 1997-07-02
          Stephen Buswell
       Revised 1997-12-15
```

```
         Stephen Buswell
      Revised 1998-02-08
         Stephen Buswell
      Revised 1998-04-04
         Stephen Buswell
      Entities and small revisions 1999-02-21
         David Carlisle
      Added attribute definitionURL to ci and cn 1999-10-11
         Nico Poppelier
      Additions for MathML 2  1999-12-16
         David Carlisle
      Namespace support 2000-01-14
         David Carlisle
      XHTML Compatibility 2000-02-23
         Murray Altheim
      New content elements 2000-03-26
         David Carlisle

-->

<!-- end of MathML 2.0 DTD  ............................................... -->
<!-- .................................................................... -->
```

Appendix B

Content Markup Validation Grammar

```
Informal EBNF grammar for Content Markup structure validation
============================================================
// Notes
//
// This defines the valid expression trees in content markup
//
// ** it does not define attribute validation -
// ** this has to be done on top
//
// Presentation_tags is a placeholder for a valid
// presentation element start tag or end tag
//
// #PCDATA is the XML parsed character data
//
// symbols beginning with '_' for example _mmlarg are internal symbols
// (recursive grammar usually required for recognition)
//
// all-lowercase symbols for example 'ci' are terminal symbols
// representing MathML content elements
//
// symbols beginning with Uppercase are terminals
// representating other tokens
//
// revised sb 3.nov.97, 16.nov.97 and 22.dec.1997
// revised sb 6.jan.98, 6.Feb.1998 and 4.april.1998
// whitespace  definitions  including presentation_tags
Presentation_tags ::= "presentation"       //placeholder
Space    ::= #x09 | #xoA | #xoD | #x20      //tab, lf, cr, space characters
S        ::= (Space | Presentation_tags)*   //treat presentation as space
// only for content validation
// characters
Char     ::= Space | [#x21 - #xFFFD]
             | [#x00010000 - #x7FFFFFFF]   //valid XML chars
```

```
// start and end tag functions
// start(\%x) returns a valid start tag for the element \%x
// end(\%x) returns a valid end tag for the element \%x
// empty(\%x) returns a valid empty tag for the element \%x
//
// start(ci)    ::= "<ci>"
// end(cn)              ::= "</cn>"
// empty(plus)  ::= "<plus/>"
//
// The reason for doing this is to avoid writing a grammar
// for all the attributes. The model below is not complete
// for all possible attribute values.
_start(\%x)      ::= "<\%x" (Char - '>')* ">"
// returns a valid start tag for the element \%x
_end(\%x)        ::= "<\%x" Space* ">"
// returns a valid end tag for the element \%x
_empty(\%x)      ::= "<\%x" (Char - '>')* "/>"
// returns a valid empty tag for the element \%x
_sg(\%x)         ::= S _start(\%x)
// start tag preceded by optional whitespace
_eg(\%x)         ::= _end(\%x) S
// end tag followed by optional whitespace
_ey(\%x)         ::= S _empty(\%x) S
// empty tag preceded and followed by optional whitespace
// mathml content constructs
// allow declare within generic argument type so we can insert it anywhere
_mmlall         ::= _container | _relation | _operator | _qualifier | _other
_mmlarg         ::= declare* _container declare*
_container      ::= _token | _special | _constructor
_token          ::= ci | cn | csymbol
_special        ::= apply | lambda | reln | fn
_constructor    ::= interval | list | matrix | matrixrow | set | vector
_other          ::= condition |  declare | sep
_qualifier      ::= lowlimit | uplimit | bvar | degree | logbase
// relations
_relation       ::= _genrel | _setrel | _seqrel2ary
_genrel         ::= _genrel2ary | _genrelnary
_genrel2ary     ::= ne
_genrelnary     ::= eq | leq | lt | geq | gt
_setrel         ::=  _seqrel2ary | _setrelnary
_setrel2ary     ::=  in | notin | notsubset | notprsubset
_setrelnary     ::= subset | prsubset
_seqrel2ary     ::= tendsto
//operators
_operator       ::= _funcop | _sepop | _arithop | _calcop
                | _seqop | _trigop | _statop | _lalgop
                | _logicop | _setop
```

```
_funcop         ::= _funcop1ary | _funcopnary
_funcop1ary     ::= inverse     | ident
_funcopnary     ::= fn| compose   // general user-defined function is n-ary
// arithmetic operators
// (note minus is both 1ary and 2ary)
_arithop        ::= _arithop1ary | _arithop2ary | _arithopnary  | root
_arithop1ary    ::= abs | conjugate | exp | factorial | minus
_arithop2ary    ::= quotient | divide | minus | power | rem
_arithopnary    ::= plus | times | max | min | gcd
// calculus
_calcop         ::= _calcop1ary | log | int | diff | partialdiff
_calcop1ary     ::= ln
// sequences and series
_seqop          ::= sum | product | limit
// trigonometry
_trigop         ::= sin | cos | tan | sec | csc | cot | sinh
                    | cosh | tanh | sech | csch | coth
                    | arcsin | arccos | arctan
// statistics operators
_statop         ::= _statopnary | moment
_statopnary     ::= mean | sdev | variance | median | mode
// linear algebra operators
_lalgop         ::= _lalgop1ary | _lalgopnary
_lalgop1ary     ::= determinant | transpose
_lalgopnary     ::= selector
// logical operators
_logicop        ::= _logicop1ary | _logicopnary | _logicop2ary | _logicopquant
_logicop1ary    ::= not
_logicop2ary    ::= implies
_logicopnary    ::= and | or | xor
_logicopquant   ::= forall | exists
// set theoretic operators
_setop          ::= _setop2ary | _setopnary
_setop2ary      ::= setdiff
_setopnary      ::= union | intersect
// operator groups
_unaryop        ::=  _func1ary | _arithop1ary | _trigop | _lalgop1ary
                     | _calcop1ary | _logicop1ary
_binaryop       ::=  _arithop2ary | _setop2ary | _logicop2ary
_naryop         ::=  _arithopnary | _statopnary  | _logicopnary
                     | _lalgopnary | _setopnary | _funcopnary
_ispop          ::= int | sum | product
_diffop         ::= diff | partialdiff
_binaryrel      ::= _genrel2ary | _setrel2ary |  _seqrel2ary
_naryrel        ::= _genrelnary | _setrelnary
//separator
sep             ::=  _ey(sep)
```

```
// leaf tokens  and data content of leaf elements
// note _mdata includes Presentation constructs here.
_mdatai         ::= (#PCDATA | Presentation_tags)*
_mdatan         ::= (#PCDATA | sep | Presentation_tags)*
ci              ::=  _sg(ci) _mdatai _eg(ci)
cn              ::=  _sg(cn) _mdatan _eg(cn)
// condition -  constraints constraints. contains either
// a single reln (relation), or
// an apply holding a logical combination of relations, or
// a set (over which the operator should be applied)
condition       ::= _sg(condition)  reln | apply | set  _eg(condition)
// domains for integral, sum , product
_ispdomain      ::= (lowlimit  uplimit?)
                | uplimit
                | interval
                | condition
// apply construct
apply           ::= _sg(apply) _applybody _eg(apply)
_applybody      ::= ( _unaryop _mmlarg  )
//1-ary ops
                                | (_binaryop _mmlarg _mmlarg)
//2-ary ops
                                | (_naryop _mmlarg*)
//n-ary ops, enumerated arguments
                                | (_naryop bvar* condition _mmlarg)
//n-ary ops, condition defines argument list
                                | (_ispop  bvar? _ispdomain? _mmlarg)
//integral, sum, product
                                | (_diffop bvar* _mmlarg)
//differential ops
                                | (log logbase? _mmlarg)
//logs
                                | (moment degree? _mmlarg*)
//statistical moment
                                | (root degree? _mmlarg)
//radicals - default is square-root
                                | (limit bvar* lowlimit? condition? _mmlarg)
//limits
                                | (_logicopquant bvar+ condition? (reln | apply))
//quantifier with explicit bound variables
// equations and relations - reln uses lisp-like syntax (like apply)
// the bvar and condition are used to construct a "such that" or
// "where" constraint on the relation
reln            ::= _sg(reln) _relnbody _eg(reln)
_relnbody       ::= ( _binaryrel bvar* condition? _mmlarg _mmlarg )
                    | ( _naryrel bvar* condition? _mmlarg*  )
// fn construct
```

```
fn                      ::= _sg(fn) _fnbody _eg(fn)
_fnbody         ::=  Presentation_tags | container
// lambda construct     - note at least 1 bvar must be present
lambda          ::=  _sg(lambda) _lambdabody _eg(lambda)
_lambdabody     ::=  bvar+ _container  //multivariate lambda calculus
//declare construct
declare         ::= _sg(declare) _declarebody _eg(declare)
_declarebody    ::= ci (fn | constructor)?
// constructors
interval        ::=  _sg(interval) _mmlarg _mmlarg _eg(interval)
//start, end define interval
set         ::=  _sg(set) _lsbody _eg(set)
list        ::=  _sg(list) _lsbody _eg(list)
_lsbody         ::=  _mmlarg*                 //enumerated arguments
                 | (bvar* condition _mmlarg)   //condition constructs arguments
matrix          ::= _sg(matrix) matrixrow* _eg(matrix)
matrixrow       ::= _sg(matrixrow) _mmlall* _eg(matrixrow)
//allows matrix of operators
vector          ::= _sg(vector) _mmlarg* _eg(vector)
//qualifiers - note the contained _mmlarg could be a reln
lowlimit        ::= _sg(lowlimit) _mmlarg  _eg(lowlimit)
uplimit         ::= _sg(uplimit) _mmlarg _eg(uplimit)
bvar            ::= _sg(bvar) ci degree? _eg(bvar)
degree          ::= _sg(degree) _mmlarg _eg(degree)
logbase         ::= _sg(logbase) _mmlarg _eg(logbase)
//relations and operators
// (one declaration for each operator and relation element)
_relation       ::= _ey(\%relation)        //for example <eq/>  <lt/>
_operator       ::= _ey(\%operator)        //for example <exp/> <times/>
//the top level math element
math            ::= _sg(math) mmlall* _eg(math)
```

Appendix C

Content Element Definitions

C.1 About Content Markup Elements

The primary role of MathML content elements is to provide a mechanism for recording the fact that a particular notational structure has a particular mathematical meaning. To this end, every content element must have a mathematical definition associated with it in some form. The purpose of this appendix is to provide *default* definitions. (An index to the definitions is provided later in this document.) The author may adapt the notation to their own particular needs by using the "definitionURL" to override these default definitions for selected content elements.

The mathematical definitions are not restricted to any one format. There are several reasons for allowing this flexibility, nearly all derived from the fact that if it is extremely important to allow authors to make use of existing definitions from the mathematical literature.

1. There is no suitable notation in common use. For example, the mathematical libraries of even the most extensive mathematical computation systems in use today capture only a small fraction of the mathematical liturature and even then, not all of mathematics is computational.
2. In most cases, the translation of a mathematical definition into a new denotational language is an inappropriate use of an author's energy and time.
3. The task of designing a new machine readable language suitable for recording semantic descriptions is an onerous one that goes substantially beyond the scope of this particular recommendation. It also overlaps substantially with efforts groups such as the OpenMath Consortium. (See also: North American OpenMath Initiative, and The European OpenMath Consortium)

The key issues for both archival and computational purposes is that there be a definition and that the author have a mechanism to specify which definition is to be used for a given instance of a notational construct. This denotational requirement is important without regard to the existence of an implementation of a particular concept or object in a mathematical computation system. The definition may be as vague as claiming that, say F, is an unknown, but differentiable function from the real numbers to the real numbers, or as complicated as requiring that F to be an elaborate function or operation as defined in some recent (or classical) research paper. The important thing is that the reader always have a way of determining how the notation is being used.

C.1.1 The Default Definitions

An author's decision to use content elements is a decision to used defined objects. In order to make this task less onerous, default definitions are provided. In this way, an author only needs to provide explicit definitions where their usage differs from the default usage.

Where possible the default definitions have been chosen to reflect common usage in the same way that most well written mathematical communications (in any format) benefit substantially from the author's use of widely used and understood terms.

Definitions are overridden in a particular instance by making use of the `definitionURL` attribute. The format of the content of that URL is unspecified. It may even be the case that the definitionURL is just a name invented by the author in which case it serves to warn the reader (and computational systems) that the author is using their own definition. It may be the URL of a mathematical paper whose whole purpose is to define a new operator, or even a reference to a traditional text in which the construct is defined. If the author's mathematical operator matches exactly with an operator in a particular computational system, an appropriate definition might be in terms of a MathML `semantics` element establishing a correspondence between two encodings. Whatever format is chosen, the only requirement is that some sort of definition be indicated.

This rest of this appendix provides detailed descriptions of the default semantics associated with each of the MathML content elements. Since this is exactly the role intended for the encodings under development by the OpenMath Consortium and one of our goals is to foster international cooperation in such standardization efforts we have presented the default definitions in a format modeled on OpenMath's *content dictionaries*. While the actual details differ somewhat from the OpenMath specification, the underlying principles are the same.

C.1.2 The Structure of an MMLdefinition.

Each MathML element is described using an XML format. The top element is `MMLdefinition`. The sub-elements identify the various parts of the description and include:

name `PCDATA` providing the name of the MathML element.

description A CDATA description of the object that an element represents. This will often include cross-references to more traditional texts or papers or existing papers on the Web.

classification Each MathML element must be classified according to its mathematical role.

- **punctuation** Some elements exist simply as an aid to parsing. For example the `sep` element is used to separate the `CDATA` defining a rational number into two parts in a manner that is easily parsed by an XML application. These objects are refered to as *punctuation*.
- **descriptor** Some elements exist simply to modify the properties of an existing element or mathematical object. For example the `declare` construct is used to reset the default attribute values, or to associate a name with a specific instance of an object. These kinds of elements are referred to as *descriptors* and the type of the resulting object is the same as that of element being modified, but with the new attribute values. No signature is provided for descriptors.
- **constructor** The remaining objects that 'contain' sub-elements are all object *constructors* of some sort or another. They combine the sub-elements into a compound mathematical object such as a constant, set, list, or a function application. For example, the `lambda` element *constructs* a function definition from a list of variables and an expression. while the `apply` element constructs a *function application.* By *function application* we mean the result of applying the first element of the apply (the function) to the zero or more remaining elements (the arguments). A *function application* represents an object in the range of the function. For each given combination of type and order of XML children, the signature of a constructor indicates the type (and sometimes subtype) of the resulting object.
- **function (operator)** The traditional mathematical functions and operators are represented in MathML by empty XML elements such as `plus` and `sin`. These *function* definitions are parameterized by their XML attribute values (for example, they may be of type vector) and are either used as is, for example when discussing the properties of a particular function or operator, or they are *applied* to arguments using the `apply`. The latter case is referred to as function application. Functions are classified according to how they are used. For example the empty `sin` element represents the *unary* mathematical function sine. The `plus` element is an *nary* operator. The *signature* of a function (see below) describes how it is to be used a mathematical function inside an `apply` element. Each combination of types of function arguments used inside an `apply` gives rise to an `apply` element of a given type.

MMLattribute Some of the XML attributes of a MathML content element have a direct impact on the mathematical semantics of the object. For example the `type` attribute of the `cn` element is used to determine what type of constant (integer, real, etc.) is being constructed. Only those attributes that affect the mathematical properties of an object are listed here and typically these also appear explicitly in the signature.

signature The signature is a systematic representation that associates the types of different possible combinations of attributes and function arguments to type of mathematical object that is constructed. The possible combinations of parameter and argument types (the left-hand side) each result in an object of some type (the right-hand side). In effect, it describes how to resolve operator overloading. For constructors, the left-hand side of the signature describes the types of the child elements and the right-hand side describes the type of object that is constructed. For functions, the left-hand side of the signature indicates the types of the parameters and arguments that would be expected when it is applied, or used to construct a relation, and the right-hand side represents the mathematical type of the object constructed by the `apply`. Modifiers modify the attributes of an existing object. For example, a *symbol* might become a *symbol of type vector*. The signature must be able to record specific attribute values and argument types on the left, and parameterized types on the right.. The syntax used for signatures is of the general form:

```
[<attribute name>=<attributevalue>]( <list of argument types> )
--> <mathematical result type>(<mathematical subtype>)
```

The MMLattributes, if any, appear in the form <name>=<value>. They are separated notationally from the rest of the arguments by square braces. The possible values are usually taken from an enumerated list, and the signature is usually affected by selection of a specific value. For the actual function arguments and named parameters on the left, the focus is on the mathematical types involved. The function argument types are presented in a syntax similar to that used for a DTD, with the one main exception. The types of the named parameters appear in the signature as <elementname>=<type> in a manner analogous for that used for attribute values. For example, if the argument is named (e.g. bvar) then it is represented in the signature by an equation as in:

```
[<attribute name>=<attributevalue>]( bvar=symbol,<argument list> ) -->
<mathematical result type>(<mathematical subtype>)
```

No mathematical evaluation ever takes place in MathML. Every MathML content element either refers to a defined object such as a mathematical function or it combines such objects in some way to build a new object. For purposes of the signature, the constructed object represents an object of a certain type parameterized type. For example the result of applying plus to arguments is an expression that represents a sum. The type of the resulting expression depends on the types of the operands, and the values of the MathML attributes.

example Examples of the use of this object in MathML are included in these elements.

property This element describes the mathematical properties of such objects. For simple associations of values with specific instances of an object, the first child is an instance of the object being defined. The second is a value or approx (approximation) element that contains a MathML description of this particular value. More elaborate conditions on the object are expressed using the MathML syntax.

comment These elements contain only PCDATA and can occur as a child of the MMLDefinition at any point.

C.2 Definitions of MathML Content Elements

C.2.1 Leaf Elements

C.2.1.1 cn

```
<MMLdefinition>
<name> cn </name>
<description>
A numerical constant.  The mathematical type of number
is given as an attribute.  The default type is "real".
Numbers such as rational, complex or real, require two
parts for a complete specification.  The parts of such
a number are separated by an empty "sep" element.
There are a number of pre-defined constants including:
      &pi; &Exponential; &ComplexI &true; &false; &NaN;
the properties of some of which are outlined below.
The &NaN; is IEEE's "Not a Number", as defined in
IEEE 854 standard for Floating point arithmetic.
</description>
<functorclass> constant </functorclass>
<MMLattribute>
    <name> type </name>
    <value> integer | rational | complex-cartesian
             | complex-polar | real
    </value>
    <default> real </default>
</MMLattribute>
<MMLattribute>
    <name> base </name>
```

```
    <value> positive_integer </value>
    <default> 10 </default>
</MMLattribute>
<signature> [type=integer](numstring) -> constant(integer) </signature>
<signature> [base=basevalue](numstring) -> constant(integer) </signature>
<signature> [type=rational](numstring,numstring) -> constant(rational) </signature>
<signature> [type=complex-cartesian](numstring,numstring) -> constant(complex) </signature>
<signature> [type=rational](numstring,numstring) -> constant(rational) </signature>
<signature> [type=real](&pi;) -> constant(real) </signature>
<signature> [definition](numstring,numstring) -> constant(userdefined) </signature>
<signature> (&gamma;) -> constant</signature>
<example> <cn> 245 </cn> </example>
<example> <cn type="integer"> 245 </cn> </example>
<example> <cn type="integer" base="16"> A </cn></example>
<example> <cn type="rational"> 245 <sep> 351 </cn> </example>
<example> <cn type="complex-cartesian"> 1 <sep/> 2 </cn> </example>
<example> <cn> 245 </cn> </example>
<property> <approx>
  <cn> &pi; </cn>
  <cn> 3.141592654 </cn>
</approx></property>
<property> <approx>
  <cn> &gamma; </cn>
  <cn> .5772156649 </cn>
</approx> </property>
<property> <reln><identity/>
  <cn>&ImaginaryI; </cn>
  <apply><root><cn>-1</cn><cn>2</cn></apply>
</reln>
</property>
<property> <reln><approx>
<cn> &ExponentialE; </cn><cn>2.718281828 </cn>
</reln> </property>
<property> <apply><forall/>
  <bvar><ci type=boolean>p</ci></bvar>
 apply><and/>
    <ci>p</ci><cn>&true;</cn></apply>
    <ci>p</ci>
  </apply>
</property>
<property> <apply><forall/>
  <bvar><ci type=boolean>p</ci></bvar>
  <apply><or/>
    <ci>p</ci><cn>&true;</cn></apply>
    <cn>&true;</cn>
  </apply>
</property>
```

```
  <bvar><ci type=boolean>p</ci></bvar>
  <apply><or/>
    <ci>p</ci><cn>&true;</cn></apply>
    <cn>&true;</cn>
  </apply>
</property>
<property>
    <identity>
      <apply><not/><cn> &true; </apply>
      <cn> &false; </cn>
    </identity>
</property>
<property> <reln><identity/>
  <cn base="16"> A </cn> <cn> 10 </cn> </reln> </property>
<property> <reln><identity/>
  <cn base="16"> B </cn> <cn> 11 </cn> </reln></property>
<property> <reln><identity/>
  <cn base="16"> C </cn> <cn> 12 </cn> </reln></property>
<property> <reln><identity/>
  <cn base="16"> D </cn> <cn> 13 </cn> </reln></property>
<property> <reln><identity/>
  <cn base="16"> E </cn> <cn> 14 </cn> </reln></property>
<property> <reln><identity/>
  <cn base="16"> F </cn> <cn> 15 </cn> </reln></property>
</MMLdefinition>
```

C.2.1.2 ci

```
<MMLdefinition>
<name> ci </name>
<description>
A symbolic name constructor. The type attribute can
be set to any valid MathML type.
</description>
<functorclass> constructor , unary </functorclass>
<MMLattribute>
  <name> type </name>
  <value> constant | matrix | set | vector | list | MathMLtype </value>
  <default> real </default>
</MMLattribute>
<signature> ({string|mmlpresentation}) -> symbol(constant) </signature>
<signature> [type=MathMLType]({string|mmlpresentation}) -> symbol(MathMLType) </signature>
<example><ci> xyz </ci> </example>
<example><ci> type="vector"> V </ci> </example>
</MMLdefinition>
```

C.2.2 Basic Content Element

C.2.2.1 apply

```
<MMLdefinition>
<name> apply </name>
<description>
This is the MathML constructor for function application.
The first argument is applied to the remaining arguments.
It may be the case that some of the child elements are
named elements. (See the signature.)
</description>
<functorclass> constructor , nary </functorclass>
<signature> (function,anything*) -> application </signature>
<example><apply><plus/><ci>x</ci><cn>1</cn></apply></example>
<example><apply><sin/><ci>x</ci></apply></example>
</MMLdefinition>
```

C.2.2.2 reln

```
<MMLdefinition>
<name> reln </name>
<description>
This is the MathML constructor for expressing a relation between
two or more mathematical objects.  The first argument indicates
the type of "relation" between the remaining arguments. (See the signature.)
No assumptions are made about the truth value of such a relation.
Typically, the relation is used as a component in the construction
of some logical assertion.   Relations may be combined into
sets, etc. just like any other mathematical object.
</description>
<functorclass> constructor </functorclass>
<signature> (function,anything*) -> reln </signature>
<example><reln><and/><ci>P</ci><ci>Q</ci></reln></example>
<example><reln><lt/><ci>x</ci><ci>y</ci></reln></example>
</MMLdefinition>
```

C.2.2.3 fn

```
<MMLdefinition>
<name> fn </name>
<description>
This is the MathML constructor for building new function
names.   The "name" can be a general MathML content element.
It identifies that object as "usable" in a function
context.
By setting its definitionURL value, you can
associate it with a particular function definition.
Use the MathML Declare to associate a name with a lambda
construct.
</description>
<MMLattribute>
  <name>definitionURL</name>
  <value> URL </value>
  <default> none </default>
</MMLattribute>
<functorclass> constructor </functorclass>
<signature> (anything) -> function </signature>
<signature> [definitionURL=functiondef](anything) ->
  function(definitionURL=functiondef)
</signature>
<example><fn><ci>F</ci></fn></example>
<example><fn definitionURL="http://www.w3c/...">
  <lt/><ci>G</ci></fn>
</example>
<!--Declaring Id to be the identity function.-->
<example>
  <declare><fn><ci>Id</ci></fn><lambda><ci>x</ci><ci>x</ci></declare>
</example>
</MMLdefinition>
```

C.2.2.4 interval

```
<MMLdefinition>
<name> interval </name>
<description>
This is the MathML constructor element for building an interval
on the real line.  While an interval could be expressed by
combining relations appropriately, they occur explicitly because
of their frequence of occurrence in common use.
</description>
<MMLattribute>
  <name>type</name>
  <value> closed | open | open-closed | closed-open </value>
  <default> closed </default>
</MMLattribute>
<functorclass> constructor , binary </functorclass>
<signature> [type=intervaltype](expression,expression) -> interval </signature>
<example><reln><and/><ci>x</ci><cn>1</cn></reln></example>
<example><reln><lt/><ci>x</ci></reln></example>
</MMLdefinition>
```

C.2.2.5 inverse

```
<MMLdefinition>
<name> inverse </name>
<description>
This MathML element is applied to a function in order to
construct a new function that is to be interpreted as the
inverse function of the original function.  For a particular
function F,  inverse(F) composed with F behaves like the
identity map on the domain of F and F composed with inverse(F)
should be an identity function on a suitably restricted
subset of the Range of F.
The MathML definitionURL attribute should be used to resolve
notational ambiguities, or to restrict the inverse to a
particular domain or make it one-sided.
</description>
<MMLattribute>
  <name>definitionURL</name>
  <value> CDATA </value>
  <default> none </default>
<!--none corresponds to using the default MathML definition ...-->
</MMLattribute>
<functorclass> operator, unary </functorclass>
<signature> (function) -> function </signature>
<signature> [definitionURL=URL](function) ->
    function(definition) </signature>
<example><apply><inverse/><sin/></apply></example>
<example>
```

```
  <apply>
  <inverse definitionURL="www.w3c.org/MathML/Content/arcsin"/>
  <sin/>
  </apply>
</example>
<property><apply><forall/>
  <bvar><ci>y</ci></bvar>
  <apply><sin/>
    <apply>
      <apply><inverse/><sin/></apply>
      <ci>y</ci>
    </apply>
  </apply>
  <value><ci>y</ci></value>
</apply>
</property>
<property>
<apply>
  <apply><inverse/><sin/></apply>
  <apply>
    <sin/>
    <ci>x</ci>
  </apply>
</apply>
<value><ci>x</ci></value>
</property>
<property>F(inverse(F)(y))<value>y</value></property>
</MMLdefinition>
```

C.2.2.6 sep

```
<MMLdefinition>
<name> sep </name>
<description>
This is the MathML infix constructor used to sub-divide PCDATA into
separate components. for example, this is used in the description of
a multipart number such as a rational or a complex number.
</description>
<functorclass> punctuation </functorclass>
<example><cn type="complex-polar">123<sep/>456</cn></example>
<example><cn>123</cn></example>
</MMLdefinition>
```

C.2.2.7 condition

```
<MMLdefinition>
<name> condition </name>
<description>
This is the MathML constructor for building conditions.
A condition differs from a relation in how it is used.
A relation is simply an expression, while a condition
is  used as a predicate to place a conditions on a bound
variables.
For a compound condition use relations or apply
operators such as "and" or "or" or a set of
relations).
</description>
<functorclass> constructor, unary </functorclass>
<signature> ({reln|apply|set}) -> predicate </signature>
<example>
<condition>
  <reln><lt/>
    <apply><power/>
      <ci>x</ci><cn>5</cn>
    </apply>
    <cn>3</cn>
  </reln>
</condition>
</example>
</MMLdefinition>
```

C.2.2.8 declare

```
<MMLdefinition>
<name> declare </name>
<description>
This is the MathML constructor for redefining the properties and
values with mathematical objects.  For example V may be a name
delcared to be a vector, or V may be a name that stands for a
particular vector.
The attribute values of the declare statement are assigned as the
corresponding default attribute values of the first object.
</description>
<functorclass> modifier , (unary | binary) </functorclass>
<MMLattribute>
<name>definitionURL</definition>
<value> Any valid URL </value>
</MMLattribute>
<MMLattribute>
<name>type</name><value> MathMLType </value>
</MMLattribute>
<MMLattribute>
<name>nargs</name><value> number of arguments for an object of type fn </value>
</MMLattribute>
<signature> [attributename=attributevalue](anything)   -> anything(attributevalue) </signature>
<!-- The two argument form updates  the properties of the first
object to be those of the second.  The attribute values override the
properties of the "value".
-->
<signature> [attributename=attributevalue](anything,anything)   -> anything(attributevalue) </signature>
<example><reln><and/><ci>x</ci><cn>1</cn></reln></example>
<example><reln><lt/><ci>x</ci></reln></example>
</MMLdefinition>
```

C.2.2.9 `lambda`

```
<MMLdefinition>
<name> lambda </name>
<description> The operation of lambda calculus that makes a
function from an expression and a variable.  The definition
at this level uses only one variable.  Lambda is a binary
function, where the first argument is the variable and
the second argument is a the expression.
Lambda( x, F ) is written as \lambda x [F] in the lambda
calculus literature.
The lambda function can be viewed as the inverse of function
application.
Although the expression F may contain x, the lambda expression
is interpreted to be free of x.  That is, the x variable is
a variable local to the environment of the definition of
the function or operator.  Formally, lambda(x,F) is free of
x, and any substitutions, evaluations or tests for x in
lambda(x,F) should not happen.
A lambda expression on an arbitrary function applied to a
simple argument is equivalent to the arbitrary function.
E.g.  lambda(x, f(x)) == f.  This is a common shortcut.
</description>
  <functorclass> Nary , Constructor </functorclass>
  <property>
    <lambda><ci>x</ci>
      <apply><fn><ci>F</ci></fn><ci>x</ci></apply>
    </lambda>
     <value> <fn><ci>F</ci></fn> </value>
  </property>
<!-- Constructing a variant of the sine function -->
   <example>
      <lambda>
        <ci> x </ci>
        <apply><sin/>
          <apply><plus/>
            <ci> x </ci>
            <cn> 3 </cn>
        </apply>
      </lambda>
   </example>
<!-- the identity operator  -->
  <example>
    <lambda><ci> x </ci> <ci> x </ci> </lambda>
  </example>
  <property>
  <reln><identity/>
    <lambda><ci>x</ci>
      <apply><fn><ci>F</ci></fn><ci>x</ci></apply>
```

```
    </lambda>
    <fn><ci>F</ci></fn>
  </reln>
  </property>
<MMLdefinition>
```

C.2.2.10 compose

```
<MMLdefinition>
<name> compose </name>
<description>
  This is the MathML constructor for composing functions.
  In order for a composition to be meaningful, the range of
  the first function must be the domain of the second function,
  etc.  .
  The result is a new function whose domain is the domain of
  the first function and whose range is the range of the last
  function and whose definition is equivalent to applying
  each function to the previous outcome in turn as in:
  (f @ g )( x )   ==  f( g(x) ).
This function is often denoted by a small circle infix
operator.
</description>
<functorclass> Nary , Operator </functorclass>
<signature> (fn*) -> fn </signature>
<example>
<apply><compose/>
       <fn><ci> f </ci></fn>
       <fn><ci> g </ci></fn>
     </apply></example>
<property>
<apply><forall>
  <bvar><ci>x</ci></bvar>
  <reln><eq/>
    <apply>
      <apply><compose/>
        <ci>f</ci>
        <ci>g</ci>
      </apply>
      <ci>x</ci>
    </apply>
    <apply><ci>f</ci>
      <apply><ci>g</ci>
        <ci>x</ci>
      </apply>
    </apply>
  </reln>
```

```
</apply>
</property>
</MMLdefinition>
```

C.2.2.11 ident

```
<MMLdefinition>
<name> ident </name>
<description>
  This is the MathML constructor for the identity function.
  This function has the property that
       f( x ) = x,  for all x in its domain.
</description>
<functorclass> Nary , Operator </functorclass>
<signature> (symbol) -> symbol </signature>
<example>
<apply><ident/>
       <ci> f </ci>
       <ci> x </ci>
</apply>
</example>
<property>
<apply><forall>
  <bvar><ci>x</ci></bvar>
  <reln><eq/>
    <apply><ident/>
        <ci>f</ci>
        <ci>x</ci>
    </apply>
    <ci>x</ci>
    </reln>
</apply>
</property>
</MMLdefinition>
```

C.2.3 Arithmetic, Algebra and Logic

C.2.3.1 quotient

```
<MMLdefinition>
The binary function used to represent
the quotient of two integers.
division.  For arguments a and b, such that
sign of a, its value would be q.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>integer</default>
</MMLattribute>
<signature> (integer, integer) -> integer </signature>
<signature> [type=integer](symbolic, symbolic) -> symbolic </signature>
<property><apply><forall/>
<bvar><ci>a</ci>
</bvar>
<bvar><ci>b</ci>
</bvar>
<reln/>
<eq/>
<ci>a</ci>
<apply><plus/>
<apply><times/>
<ci>b</ci>
<apply><quotient/>
<ci>a</ci>
<ci>b</ci>
</apply>
</apply>
<apply><rem/>
<ci>a</ci>
<ci>b</ci>
</apply>
</apply>
<apply/>
</apply></property>
<property><apply><ident/>
<apply><quotient/>
<ci>5</ci>
<ci>4</ci>
```

```
</apply>
<ci>1</ci>
</apply></property>
=======
<name> quotient </name>
    <description> Integer quotient, the result of integer
      division.  For arguments a and b, it returns q,
      where a = b*q+r,  |r| &lt; |b|  and  a*r &gt;= 0  (or
      the sign of r is the same as the sign of a).
     </description>
       <functorclass> Binary, Function </functorclass>
       <signature> (integer, integer) -> integer </signature>
       <signature> (symbolic, symbolic) -> symbolic  ->  =>  &#8594; </signature>
       <property>
       <description>
       ForAll(bvar(a,b),identity(a ,b*Quotient(a,b) + Remainder(a,b))
       </description>
       <apply><forall/>
         <bvar><ci>a</ci></bvar>
         <bvar><ci>b</ci></bvar>
         <reln/><eq/>
             <ci>a</ci>
             <apply><plus/>
               <apply><times/>
                 <ci>b</ci>
                 <apply><quotient/><ci>a</ci><ci>b</ci></apply>
               </apply>
               <apply><rem/><ci>a</ci><ci>b</ci></apply>
             </apply>
         <reln>
       </apply>
       </property>
       <property>
     <description>
   1 = quotient(5,4)
     </description>
      <apply><identity/>
        <apply><quotient/>
          <ci>5</ci>
          <ci>4</ci>
        </apply>
        <ci>1</ci>
     <apply>
     </property>
</MMLdefinition>
```

C.2.3.2 exp

```
<MMLdefinition>
The exponential function.
<reference> M. Abramowitz and I. Stegun, Handbook of
Mathematical Functions, [4.2]</reference>
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
any MathML Type
</values>
<default>real</default>
</MMLattribute>
<signature> real -> real </signature>
<signature> symbolic -> symbolic </signature>
<property><apply><eq/>
<apply><exp/>
<cn>0</cn>
</apply>
<cn>1</cn>
</apply></property>
<property><apply><ident/>
<apply><exp/>
<ci>x</ci>
</apply>
<apply><power/>
<cn>ExponentialE;</cn>
<ci>x</ci>
</apply>
</apply></property>
<property> exp(x) = limit( (1+x/n)^n, n, infinity ) </property>
</MMLdefinition>
```

C.2.3.3 factorial

```
<MMLdefinition>
This element is used to construct factorials
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
any MathML Type
</values>
<default>integerl</default>
</MMLattribute>
<signature> ( algebraic ) -> algebraic </signature>
<signature>(integer)->integer</signature>
<property><apply><forall/>
<bvar><ci>n</ci></bvar>
<condition><apply><gt/>
<ci>n</ci>
<cn>0</cn>
</apply>
</condition>
<apply><eq/>
<apply><factorial/><ci>n</ci></apply>
  <apply><times/>
    <ci>n</ci>
    <apply><factorial/>
      <apply><minus/><ci>n</ci><cn>1</cn></apply>
    </apply>
  </apply>
</apply>
</apply>
</property>
<example><apply><factorial/>
<ci>n</ci>
</apply></example>
</MMLdefinition>
```

C.2.3.4 divide

```
<MMLdefinition>
This is the binary  MathML operator that is used to construct
the mathematical expression  a "divided by" b. In
general, it constructs the expression that
is equivalent to right multiplication by
the multiplicative inverse of b.
classification=function
<MMLattribute>
<attname> type </attname>
<attvalue> anything <sep/>non-commutative</attvalue>
<attdefault> real </attdefault>
</MMLattribute>
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<signature> (complex, complex) -> complex </signature>
<signature> (real, real) -> real </signature>
<signature> (rational, rational) -> rational </signature>
<signature> (integer, integer) -> rational </signature>
<signature> (symbolic, symbolic) -> symbolic </signature>
<property><apply><forall/>
  <bvar><ci>a</ci></bvar>
  <apply><eq/>
    <apply><divide/>
      <ci> a </ci>
      <ci> 0 </ci>
    </apply>
    <apply><ci>Error</ci>
      <ci>Division by 0</ci>
    </apply>
  </apply>
</apply>
</property>
<property>whenever not(a=0) then a/a = 1 </property>
<example><apply><divide/>
<ci> a </ci>
<ci> b </ci>
</apply></example>
</MMLdefinition>
```

C.2.3.5 max

```
<MMLdefinition>
Represent the maximum of a set of elements.  The elements
may be listed explicitly or they may be  described by a
condition,  e.g., the maximum over all x in
the set A.
To be well defined, the elements must all be
comparable.
classification= function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
any MathML Type
</values>
<default>real</default>
</MMLattribute>
<signature> ( ordered_set_element * ) -> ordered_set_element </signature>
<signature> ( bvar,condition,anything ) ->  ordered_set_element </signature>
<example><apply><max/>
<cn>2</cn>
<cn>3</cn>
<cn>5</cn>
</apply></example>
<example><apply>
  <max/>
  <bvar><ci>y</ci></bvar>
  <condition>
  </condition>
  <apply>
      <power/>
    <ci> y</ci>
    <cn>x </cn>
  </apply>
</apply>
</example>
</MMLdefinition>
```

C.2.3.6 min

```
<MMLdefinition>
Represent the maximum of a set of elements.  The elements
may be listed explicitly or they may be  described by a
condition,  e.g., the maximum over all x in
the set A.
To be well defined, the elements must all be
comparable.
classification= function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
any MathML Type
</values>
<default>real</default>
</MMLattribute>
<signature> ( ordered_set_element * ) -> ordered_set_element </signature>
<signature> ( bvar,condition,anything ) ->  ordered_set_element </signature>
<example><apply><min/>
<cn>2</cn>
<cn>3</cn>
<cn>5</cn>
</apply></example>
<example><apply>
  <min/>
  <bvar><ci>x</ci></bvar>
  <condition>
  </condition>
  <apply>
      <power/>
    <ci> x </ci>
    <cn> 2 </cn>
  </apply>
</apply>
</example>
</MMLdefinition>
```

C.2.3.7 minus

```
<MMLdefinition>
The subtraction operator for an additive group.
```

```
If one argument is provided this constructs the additive
inverse of that group element.
If two arguments, say a and b, are provided it
constructs the mathematical expression  a - b.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
any MathML Type
</values>
<default>real</default>
</MMLattribute>
<signature>[type=typevalue](typevalue,typevaluel) -> typevalue </signature>
<signature>[type=typevalue](typevalue)->typevalue </signature>
<property><apply><eq/>
<bvar><ci>n</ci>
</bvar>
<apply><minus/>
<cn>1</cn>
</apply>
<cn>-1</cn>
</apply></property>
<example><apply><minus/>
<cn>3</cn>
<cn>5</cn>
</apply></example>
<example><apply><minus/>
<cn>3</cn>
</apply></example>
</MMLdefinition>
```

C.2.3.8 plus

```
<MMLdefinition>
The N-ary addition operator of an algebraic structure.
If no operands are provided, the expression represents
the additive identity.
If one operand, a, is provided the expression would
evaluate to "a".
If two or more operands are provided, the expression
represents the group element corresponding to a left
associative binary pairing of the operands.
Issues with regard to the "value" of mixed operands
are left up to the target system.  If the author wishes
to refer to specific type coercion rules, then
the definitionURL attribute should be used to refer
to a suitable specification.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>real</default>
</MMLattribute>
<signature>[type=typevalue](typevalue*) -> typevalue </signature>
<property> plus( ) = 0 </property>
<property> +(a) = a </property>
<property> ForAll(a,Commutative, a + b = b + a)</property>
<example><apply><plus/>
<cn>3</cn>
</apply></example>
<example><apply><plus/>
<cn>3</cn>
<cn>5</cn>
</apply></example>
<example><apply><plus/>
<cn>3</cn>
<cn>5</cn>
<cn>7</cn>
</apply></example>
</MMLdefinition>
```

C.2.3.9 power

```
<MMLdefinition>
The binary powering operator used to construct expressions
such as a "to the power of" b.  In particular, it is the
operation for which a "to the power of" 2  is equivalent
to a * a.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>real</default>
</MMLattribute>
<signature> (complex complex) -> complex </signature>
<signature> (real real) -> complex </signature>
<signature> (rational rational) -> complex </signature>
<signature> (rational integer) -> rational </signature>
<signature> (integer integer) -> rational </signature>
<signature> (symbolic symbolic) -> symbolic </signature>
<signature>[type=typevalue](typevalue,typevalue) -> typevalue </signature>
<property> ForAll(a,Condition(a_NE_0),a^0=1) </property>
<property> ForAll(a,a^1=a) </property>
<property> ForAll(a,1^a=1) </property>
<property>ForAll(a,0^0=Undefined)</property>
</MMLdefinition>
```

C.2.3.10 rem

```
<MMLdefinition>
Integer remainder, the result of integer
division.  For arguments a and b, such that
the same as the  sign of a, its value would be r.
classification= binary, function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>integer</default>
</MMLattribute>
<signature> (integer integer) -> integer </signature>
<signature> (symbolic symbolic) -> symbolic </signature>
<signature>[type=typevalue](typevalue,typevalue)->typevalue</signature>
<property> a = b*rem(a,b) + rem(a,b) </property>
<property>rem(a,0) = Division_by_Zero</property>
</MMLdefinition>
```

C.2.3.11 times

```
<MMLdefinition>
The n-ary  multiplication operator of a
ring.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>real</default>
</MMLattribute>
<signature> (complex *) -> complex </signature>
<signature> (real*) -> real </signature>
<signature> (rational*) -> rational </signature>
<signature> (integer*) -> integer </signature>
<signature> (symbolic*) -> symbolic </signature>
<property>ForAll(bvars(a,b),condition(in({a,b},Commutative)),a*b=b*a)</property>
<property>ForAll(bvars(a,b,c),Associative,a*(b*c)=(a*b)*c), associativity </property>
<property> a*1=a </property>
<property> 1*a=a </property>
<property> a*0=0 </property>
<property> 0*a=0 </property>
</MMLdefinition>
```

C.2.3.12 root

```
<MMLdefinition>
Construct the nth root of an object.
The first argument "a" is the object and the
second object "n" denotes the root, as in
( a ) ^ (1/n)
classification= binary , function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<attname> type </attname>
<attvalue> real <sep/> complex <sep/> principle_branch </attvalue>
<attdefault> real </attdefault>
</MMLattribute>
<signature> ( anything , anything) -> root </signature>
<property> Forall(bvars(a,n),root(a,n) = a^(1/n)) </property>
<example><apply><root/>
<ci> a </ci>
<ci> n </ci>
</apply></example>
</MMLdefinition>
```

C.2.3.13 gcd

```
<MMLdefinition>
This operator is used to construct an expression
which represents the greatest common divisor
of its arguments.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>integer</default>
</MMLattribute>
<signature> [type=typevalue](typevalue*) ->typevalue </signature>
<property>Forall(p,q,(is(p,prime) and is(q,prime)) , gcd(p,q)=1 </property>
<example><apply><gcd/>
<cn>12</cn>
<cn>17</cn>
</apply></example>
</MMLdefinition>
```

C.2.3.14 and

```
<MMLdefinition>
This is the n-ary  logical "and" operator.  It is used
to construct the logical expression which has
a  value of "true" when all of its operands
have a truth value of "true", and "false" otherwise.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<attname> type </attname>
<attvalue> any MathML type</attvalue>
<attdefault> complex </attdefault>
</MMLattribute>
<signature> (boolean*) -> boolean </signature>
<signature> [type="boolean"](symbolic*)      -> boolean </signature>
<property> identity(true and p , p ) </property>
<property> identity(p and q , q and p ) </property>
<example><apply><and/>
<ci>p</ci>
<ci>q</ci>
</apply></example>
</MMLdefinition>
```

C.2.3.15 or

```
<MMLdefinition>
The logical "or" operator.  The constructed expression
has a truth value of true if at least one of its arguments is true.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>boolean</default>
</MMLattribute>
<signature> (boolean*) -> boolean </signature>
<signature> [type="boolean"](symbolic*)      -> boolean </signature>
<property> ...</property>
</MMLdefinition>
```

C.2.3.16 xor

```
<MMLdefinition>
The logical "xor" operator.  The constructed expression
has a truth value of true if exactly one of its arguments is true.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>boolean</default>
</MMLattribute>
<signature> (boolean*) -> boolean </signature>
<signature> [type="boolean"](symbolic*)      -> symbolic </signature>
</MMLdefinition>
```

C.2.3.17 not

```
<MMLdefinition>
The logical "not" operator negates the truth value
of its single argument.  e.g.,  not P
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>boolean</default>
</MMLattribute>
<signature> (boolean) -> boolean </signature>
<signature> [type="boolean"](symbolic)      -> symbolic </signature>
</MMLdefinition>
```

C.2.3.18 implies

```
<MMLdefinition>
The implies operator.  This represents
the construction of the logical expression
  "A implies B".
classification= Binary, relation
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>boolean</default>
</MMLattribute>
<signature> (boolean,boolean) -> boolean </signature>
<property><apply><forall/>
<bvar><ci>A</ci>
</bvar>
<bvar><ci>B</ci>
</bvar>
<apply><eq/>
<apply><implies/>
```

```
<ci>A</ci>
<ci>B</ci>
</apply>
<apply><or/>
<ci>B</ci>
<apply><not/>
<ci> A </ci>
</apply>
</apply>
</apply>
</apply></property>
</MMLdefinition>
```

C.2.3.19 forall

```
<MMLdefinition>
The logical "For all" quantifier is applied to arguments
to construct a predicate.  The bound variables are
tagged using bvar, and the last argument is the boolean
predicate that is asserted to be true.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>boolean</default>
</MMLattribute>
<signature> (bvar*,condition?,apply) -> boolean </signature>
<signature> (bvar*,condition?,(reln)) -> boolean </signature>
</MMLdefinition>
```

C.2.3.20 exists

```
<MMLdefinition>
This is the MathML operator that is used to
assert existance, as in "There exists an x such
that x is real and x is positive."
It expects three arguments.
The first argument indicates the bound variable,
The second argument places conditions on that
bound variable.  The last argument is the expression
that is asserted to be true.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>boolean</default>
</MMLattribute>
<signature> (element,set) ->boolean </signature>
</MMLdefinition>
```

C.2.3.21 abs

```
<MMLdefinition>
A unary operator which represents the absolute value of its argument.
In the complex case this is often referred to as the modulus.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
any MathML Type
</values>
<default>real</default>
</MMLattribute>
<signature>(real)->real</signature>
<signature>(complex)->real</signature>
<property>for all x and y, abs(x) + abs(y) >= abs(x+y)
</property>
<example><apply><abs/><ci>x</ci></apply></example>
</MMLdefinition>
```

C.2.3.22 conjugate

```
<MMLdefinition>
The "conjugate" arithmetic operator is
used to represent the complex conjugate of its
argument.  In particular, conjugate( ImaginaryI )
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<attname> type </attname>
<attvalue> anything </attvalue>
<attdefault> complex </attdefault>
</MMLattribute>
<signature> (algebraic) -> algebraic </signature>
<signature>(complex)->complex</signature>
</MMLdefinition>
```

C.2.3.23 arg

```
<MMLdefinition>
The "arg" operator is used to construct an
expression which represents the
"argument" of a complex number.
classification=function
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<MMLattribute>
<name>type</name>
<values>
any MathML Type
</values>
<default>real</default>
</MMLattribute>
<signature>(compex)->real</signature>
<property>???</property>
<ci>a</cn>
<ci>&epsilon</cn>
<ci><mrow><msup><mi>a</mi><mi>b</mi><mrow></cn>
<ci>v</ci>
</MMLdefinition>
```

C.2.3.24 real

```
<MMLdefinition>
An operator used to construct an expression
representing the "real" part of a complex number.
classification=unary
<MMLattribute>
<name>type</name>
<values>
Any MathML type
</values>
<default>real</default>
</MMLattribute>
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<signature>(complex)->real</signature>
<ci>a</cn>
<ci>&epsilon</cn>
<ci><mrow><msup><mi>a</mi><mi>b</mi><mrow></cn>
<ci>v</ci>
</MMLdefinition>
```

C.2.3.25 imaginary

```
<MMLdefinition>
A name used as a symbolic identifier.
classification=constant
<MMLattribute>
<attname>definitionURL</attname>
<attvalue> CDATA </attvalue>
<attdefault> none </attdefault>
</MMLattribute>
<signature>(complex)->real</signature>
<example><cn type="constant">&Imaginary;</cn></example>
</MMLdefinition>
```

C.2.4 Relations

C.2.4.1 eq

```
<MMLdefinition>
<Name> eq </Name>
<description> The equality operator. </description>
<functorclass> Nary, relation </functorclass>
<property> Commutative </property>
<signature> (symbolic symbolic) -> boolean </signature>
</MMLdefinition>
```

C.2.4.2 neq

```
<MMLdefinition>
<Name> neq </Name>
<description> The notequals operator. </description>
<functorclass> Nary, relation </functorclass>
<property> Commutative </property>
<signature> (symbolic symbolic) -> boolean </signature>
</MMLdefinition>
```

C.2.4.3 gt

```
<MMLdefinition>
<Name> gt </Name>
<description> The equality operator. </description>
<functorclass> binary, relation </functorclass>
<property> Commutative </property>
<signature> (symbolic symbolic) -> boolean </signature>
</MMLdefinition>
```

C.2.4.4 lt

```
<MMLdefinition>
<Name> lt </Name>
<description> The inequality equality operator "<" </description>
<functorclass> binary, relation </functorclass>
<property> Commutative </property>
<signature> (symbolic, symbolic*) -> boolean </signature>
</MMLdefinition>
```

C.2.4.5 geq

```
<MMLdefinition>
<Name> geq </Name>
<description> The inequality operator. >= </description>
<functorclass> Nary, relation </functorclass>
<signature> (symbolic, symbolic*) -> boolean </signature>
<property> ... Commutative ? ... </property>
</MMLdefinition>
```

C.2.4.6 leq

```
<MMLdefinition>
<Name> leq </Name>
<description> The inequality operator  </description>
<functorclass> Nary, relation </functorclass>
<property> Commutative </property>
<signature> (symbolic symbolic) -> boolean </signature>
</MMLdefinition>
```

C.2.5 Calculus

C.2.5.1 ln

```
<MMLdefinition>
<name>ln</name>
<description>
The logarithmic function.  Also called the natural logarithm.  The inverse
of the exponential function.
<Reference> M. Abramowitz and I. Stegun, Handbook of
Mathematical Functions, [4.1]
</Reference>
</description>
<functorclass> Unary, Function </functorclass>
<property>
    Error( "logarithm has a singularity at 0" )
</property>
<signature> Intersect(real,positive) -> real </signature>
<signature> symbolic -> symbolic </signature>
<property> ln(1) = 0 </property>
<property> ln(exp(x)) = x, "for real x" </property>
<property> exp(ln(x)) = x, always </property>
</MMLdefinition>
```

C.2.5.2 log

```
    <MMLdefinition>
        <Name> log </Name>
        <description> The logarithmic function (base 10), or any
any other user specified base.  Also called
the natural logarithm.
The inverse of the exponential function.
            <Reference> M. Abramowitz and I. Stegun, Handbook of
Mathematical Functions, [4.1]
            </Reference>
        </description>
        <functorclass> Unary, Function </functorclass>
        <signature> (real,logbase) -> real </signature>
        <signature> symbolic -> symbolic </signature>
        <property>
            Error( "logarithm has a singularity at 0" )
        </property>
      </MMLdefinition>
```

C.2.5.3 int

```
<MMLdefinition>
      <Name> int </Name>
      <description>
          The definite or indefinite integral of a function or algebraic
          expression.
          There are several forms of calling sequences depending on
          the nature of the areguments, and whether or not it is a
          definite integral.
      </description>
      <functorclass> Binary , Function </functorclass>
      <signature> (function) -> function </signature>
      <signature> (algebraic,bvar) -> algebraic </signature>
      <signature> (algebraic,bvar,interval) -> algebraic </signature>
      <signature> (algebraic,bvar,condition) -> algebraic </signature>
   </MMLdefinition>
```

C.2.5.4 diff

```
<MMLdefinition>
      <Name> diff </Name>
      <description>
          For expressions, this represents the derivative of
          its first argument evaluated at the second argument.
          For Unary functions (only one argument) it represents
          f'.
      </description>
      <functorclass> (Unary | Binary) , Function </functorclass>
      <signature> (algebraic,bvar) -> algebraic </signature>
      <property>Forall(x,diff( sin(x) , x ) = cos(x)) </property>
      <property>Forall(x,diff( x , x ) = 1 ) </property>
      <property>Forall(x,diff( x^2 , x ) = 2x) </property>
      <property>identity( diff(sin) , cos ) </property>
   </MMLdefinition>
```

C.2.5.5 partialdiff

```
<MMLdefinition>
    <Name> partialdiff </Name>
    <description>
        For expressions, this represents the derivative of
        its first argument evaluated at the second argument.
        For Unary functions (only one argument) it represents
        f'.
    </description>
    <functorclass> (Binary) , Function </functorclass>
    <signature> (algebraic,bvar) -> algebraic </signature>
    <property>Forall(x,diff( sin(x*y) , x ) = cos(x)) </property>
    <property>Forall(x,y,diff( x*y , x ) = diff(x,x)*y + diff(y,x)*x ) </property>
    <property>Forall(x,a,b,diff( a + b , x ) = diff(a,x) + diff(b,x) ) </property>
    <property>identity( diff(sin) , cos ) </property>
</MMLdefinition>
```

C.2.5.6 lowlimit

```
<MMLdefinition>
    <Name> lowlimit </Name>
    <description> Construct a lower limit.  Limits
     are used in some integrals as alternative way
     of describing the region over which an integral
     is computed. (i.e. a connected component of the
     real line.)
     </description>
    <functorclass> Constructor </functorclass>
    <signature> (anything*) -> list </signature>
</MMLdefinition>
```

C.2.5.7 uplimit

```
<MMLdefinition>
    <Name> uplimit </Name>
    <description> Construct a an upper limit.  Limits
     are used in some integrals as alternative way
     of describing the region over which an integral
     is computed. (i.e. a connected component of the
     real line.)
     </description>
    <functorclass> Constructor </functorclass>
    <signature> (anything*) -> list </signature>
</MMLdefinition>
```

C.2.5.8 bvar

```
    <MMLdefinition>
        <Name> bvar </Name>
        <description>
The bvar element is the container element
for the "bound variable" of an operation.
For example, in an integral it specifies the
variable of integration. In a derivative, it
indicates which variable with respect to
which a function is being differentiated.
When the bvar element is used to quantifiy a derivative,
the bvar element may contain a child degree element that
specifies the order of the derivative with respect to that
variable. The bvar element is also used for the internal
variable in sums and products.
    </description>
        <functorclass> Constructor </functorclass>
        <signature> (symbol) -> symbol </signature>
        <example> <bvar><ci>x</ci></bvar></example>
    </MMLdefinition>
```

C.2.5.9 degree

```
    <MMLdefinition>
        <Name> degree </Name>
        <description> A parameter used by some
        MathML data-types to specify that, for example,
        a bound variable is repeated several times.
        </description>
        <functorclass> Constructor </functorclass>
        <signature> (algebraic) -> algebraic </signature>
        <example> <degree><ci>x</ci></degree></example>
        <property> ... </property>
    </MMLdefinition>
```

C.2.6 Theory of Sets

C.2.6.1 set

```
    <MMLdefinition>
        <Name> set </Name>
        <description> Construct a set. </description>
        <functorclass> Nary, Constructor </functorclass>
        <signature> (anything*) -> set </signature>
    </MMLdefinition>
```

C.2.6.2 list

```
<MMLdefinition>
    <Name> list </Name>
    <description> Construct a list. </description>
    <functorclass> Nary, Constructor </functorclass>
    <signature> (anything*) -> list </signature>
</MMLdefinition>
```

C.2.6.3 union

```
<MMLdefinition>
    <Name> union </Name>
    <description> The union of two sets. </description>
    <functorclass> Binary, Function </functorclass>
    <signature> (set*) -> set </signature>
</MMLdefinition>
```

C.2.6.4 intersect

```
<MMLdefinition>
    <Name> intersection </Name>
    <description> The intersection of two sets. </description>
    <functorclass> Binary, Function </functorclass>
    <signature> (set set) -> set </signature>
</MMLdefinition>
```

C.2.6.5 in

```
<MMLdefinition>
    <Name> in </Name>
    <description>
        The membership testing operation (also commonly
        called "in" or "including").  Returns true if the first
        argument is part of the second argument.  The second
        argument must be a set.
    </description>
    <functorclass> Binary, Function </functorclass>
    <signature> (anything, set) -> boolean </signature>
</MMLdefinition>
```

C.2.6.6 notin

```
<MMLdefinition>
<Name> notin </Name>
<description>
    The membership exclusion operation (also commonly
    called "notin" or "including").
    It is defined as "not in".
</description>
    <functorclass> Binary, Function </functorclass>
    <signature> (anything set) -> boolean </signature>
</MMLdefinition>
```

C.2.6.7 subset

```
<MMLdefinition>
<Name> subset </Name>
<description>
  Boolean function whose value is determined by
  whether or not one set is a subset of another.
 </description>
    <functorclass> Binary, Function </functorclass>
    <signature> (set*) -> boolean </signature>
</MMLdefinition>
```

C.2.6.8 prsubset

```
<MMLdefinition>
<Name> prsubset </Name>
<description>
  Boolean function whose value is determined by
  whether or not one set is a proper subset of another.
 </description>
    <functorclass> Binary, Function </functorclass>
    <signature> (set, set) -> boolean </signature>
    <property>...</property>
</MMLdefinition>
```

C.2.6.9 notsubset

```
<MMLdefinition>
<Name> notsubset </Name>
<description>
  Boolean function whose value is the complement
  of "subset".
 </description>
    <functorclass> Binary, Function </functorclass>
    <signature> (set, set) -> boolean </signature>
    <property>...</property>
</MMLdefinition>
```

C.2.6.10 notprsubset

```
<MMLdefinition>
<Name> notprsubset </Name>
<description>
  Boolean function whose value is the complement
  of "proper subset".
 </description>
    <functorclass> Binary, Function </functorclass>
    <signature> (set, set) -> boolean </signature>
    <property>...</property>
</MMLdefinition>
```

C.2.6.11 setdiff

```
<MMLdefinition>
<Name> setdiff </Name>
<description>
  Function indicating the difference of two sets.
</description>
<functorclass> Binary, Function </functorclass>
<signature> (set, set) -> set </signature>
<property>...</property>
</MMLdefinition>
```

C.2.7 Sequences and Series

C.2.7.1 sum

```
<MMLdefinition>
<Name> sum </Name>
<description>
The sum element denotes the summation operator. Upper and lower
limits for the sum, and more generally a domains for the bound variables
are specified using uplimit, lowlimit or a condition on the bound
variables. The index for the summation is specified by a bvar element.
The sum element takes the attribute definition that can be used to
override the default semantics.
</description>
<functorclass> Unary, Function </functorclass>
<signature> (bvar*,((lowlimit,uplimit)|condition),algebraic) -> sum </signature>
<signature> ... </signature>
</MMLdefinition>
```

C.2.7.2 product

```
<MMLdefinition>
<Name> product </Name>
<description>
The product element denotes the product operator. Upper and lower
limits for the product, and more generally a domains for the bound
variables are specified using uplimit, lowlimit or a condition on the
bound variables. The index for the product is specified by a bvar
element.
The product element takes the attribute definition that can be used
to override the default semantics.
</description>
<functorclass> Unary, Function </functorclass>
<signature> (bvar*,((lowlimit,uplimit)|condition),algebraic)   -> product </signature>
<signature> ... </signature>
<signature> ... </signature>
</MMLdefinition>
```

C.2.7.3 limit

```
<MMLdefinition>
<Name> limit </Name>
<description>
The sum element denotes the summation operator.
Upper and lower limits for the sum, and more
generally a domains for the bound variables are
specified using uplimit, lowlimit or a condition
on the bound variables. The index for the summation is
specified by a bvar element.
</description>
<functorclass> Nary, Function </functorclass>
<signature> (bvar*,(lowlimit | condition*),algebraic) -> limit </signature>
</MMLdefinition>
```

C.2.7.4 tendsto

```
<MMLdefinition>
<Name> tendsto </Name>
<description> tendsto is used to specify how a limit is
computed. It accepts a type attribute that determines the
manner in which it tends to a value.
</description>
<functorclass> binary, Function </functorclass>
<signature> (symbol,anything) -> condition(limit) </signature>
<signature> [type=direction](symbol,anything) ->    condition(limit) </signature>
</MMLdefinition>
```

C.2.8 Trigonometry

C.2.8.1 sin

```
<MMLdefinition>
    <Name> sin </Name>
    <description> The circular trigonometric function sine
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property> sin(0) = 0 </property>
    <property> sin(integer*Pi) = 0 </property>
    <property> sin((Z+1/2)*Pi) = (-1)^Z, "for integer Z" </property>
    <property> -1 <= sin(real) </property>
    <property> sin(real) <= 1 </property>
    <property> sin(3*x)=-4*sin(x)^3+3*sin(x), "triple angle formula"
        <Reference> ditto, [4.3.27] </Reference>
    </property>
</MMLdefinition>
```

C.2.8.2 cos

```
<MMLdefinition>
    <Name> cos </Name>
    <description> The cosine function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property> cos(0) = 1 </property>
    <property> cos(integer*Pi+Pi/2) = 0 </property>
    <property> cos(Z*Pi) = (-1)^Z, "for integer Z" </property>
    <property> -1 <= cos(real) </property>
    <property> cos(real) <= 1 </property>
</MMLdefinition>
```

C.2.8.3 tan

```
<MMLdefinition>
    <Name> tan </Name>
    <description> The tangent function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property> tan(integer*Pi) = 0 </property>
    <property> tan(x) = sin(x)/cos(x) </property>
</MMLdefinition>
```

C.2.8.4 sec

```
<MMLdefinition>
    <Name> sec </Name>
    <description> The secant function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property> sec(x) = 1/cos(x) </property>
</MMLdefinition>
```

C.2.8.5 csc

```
<MMLdefinition>
    <Name> csc </Name>
    <description> The cosecant function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property> csc(x) = 1/sin(x) </property>
</MMLdefinition>
```

C.2.8.6 cot

```
<MMLdefinition>
    <Name> cot </Name>
    <description> The cotangent function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property> cot(integer*Pi+Pi/2) = 0 </property>
    <property> cot(x) = cos(x)/sin(x) </property>
</MMLdefinition>
```

C.2.8.7 sinh

```
<MMLdefinition>
    <Name> sinh </Name>
    <description> The hyperbolic sine function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property>...</property>
</MMLdefinition>
```

C.2.8.8 cosh

```
<MMLdefinition>
    <Name> sinh </Name>
    <description> The hyperbolic sine function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property>...</property>
</MMLdefinition>
```

C.2.8.9 tanh

```
<MMLdefinition>
    <Name> tanh </Name>
    <description> The hyperbolic tangent function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property>...</property>
</MMLdefinition>
```

C.2.8.10 sech

```
<MMLdefinition>
    <Name> sech </Name>
    <description> The hyperbolic secant function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property>...</property>
</MMLdefinition>
```

C.2.8.11 csch

```
<MMLdefinition>
    <Name> csch </Name>
    <description> The hyperbolic cosecant function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property>...</property>
</MMLdefinition>
```

C.2.8.12 coth

```
<MMLdefinition>
    <Name> coth </Name>
    <description> The hyperbolic cotangent function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.3]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property>...</property>
</MMLdefinition>
```

C.2.8.13 arcsin

```
<MMLdefinition>
    <Name> arcsin </Name>
    <description> The inverse of the sine function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.4]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property> sin(arcsin(x)) = x </property>
    <property> arcsin(sin(x)) = x, "for x between -Pi/2 and Pi/2" </property>
</MMLdefinition>
```

C.2.8.14 arccos

```
<MMLdefinition>
    <Name> arccos </Name>
    <description> The inverse of the cosine function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.4]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property> cos(arccos(x)) = x </property>
    <property> arccos(cos(x)) = x, "for x between 0 and Pi" </property>
</MMLdefinition>
```

C.2.8.15 arctan

```
<MMLdefinition>
    <Name> arctan </Name>
    <description> The inverse of the tangent function.
        <Reference> M. Abramowitz and I. Stegun, Handbook of
            Mathematical Functions, [4.4]
        </Reference>
    </description>
    <functorclass> Unary, Function </functorclass>
    <signature> real -> real </signature>
    <signature> symbolic -> symbolic </signature>
    <property> tan(arctan(x)) = x </property>
    <property> arctan(tan(x)) = x, "for x between -Pi/2 and Pi/2" </property>
</MMLdefinition>
```

C.2.9 Statistics

C.2.9.1 mean

```
<MMLdefinition>
  <Name> mean </Name>
  <description>
    Given  k unspecified scalar arguments they are treated as equiprobable
    values of a random variable and the mean is computed as:
      mean( a1, a2, ...  an)   Sum( ai, i=1... n )/ n.
    (see section 7.7 in CRC's Standard Mathematical tables and Formulae).
    More generally, if the first argument is a symbol X of type
    "discrete_random_variable", this is the 1st moment of the
    random variable X and is defined as
    E[ X ] = Sum( x*f(x), x in S )
    where the probability that x = x_i is  P( x = x_i) = f(x_i) .
    The arguments are either all data, all discrete random variables,
    or all continuous random variables.
    The generalizes to continuous distributions and
    k dimenions following the definitions provided in the reference:
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996, [7.1.2] and [7.7]
    </Reference>
  </description>
  <MMLattribute>
    <name>type</name>
    <values> random_variable | continuous_random_variable | data </value>
    <default> data </default>
  </MMLattribute>
  <functorclass>Nary , Operator </functorclass>
  <signature>(scalar*) -> scalar</signature>
  <signature>(scalar(type=data)*) -> scalar</signature>
  <signature>(symbol(type=random_variable)*) -> scalar</signature>
  <signature>(symbol(type=continuous_random_variable)*) -> scalar</signature>
```

```
  <property> </property>
</MMLdefinition>
```

C.2.9.2 sdev

```
<MMLdefinition>
  <Name> sdev </Name>
  <description>
    This represents the standard deviation.
    Given  k unspecified scalar arguments they are treated as equiprobable
    values of a random variable and the "standard deviation" is
    computed as the square root of the second moment about the mean  U.
      sdev( a1, a2, ...  an)^2  = E( (X - U)^2 ).
    If the first argument is a symbol X of type
    "discrete_random_variable", then all arguments are treated as
    discrete random variables, instead of data and the second moment
    about the mean is computed as
      Sum( ( x_i - U )^2 * f(x_i) , x_i  in S )
    as
    where the probability that x = x_i is  P( x = x_i) = f(x_i) .
    The arguments are either all data, all discrete random variables,
    or all continuous random variables.
    The generalizes to continuous distributions and to
    k dimenions following the definitions found in:
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996, [7.1.2] and [7.7]
    </Reference>
  </description>
  <MMLattribute>
    <name>type</name>
    <values> random_variable | continuous_random_variable | data </value>
    <default> data </default>
  </MMLattribute>
  <functorclass>Nary , Operator </functorclass>
  <signature>(scalar*) -> scalar</signature>
  <signature>(scalar(type=data)*) -> scalar</signature>
  <signature>(symbol(type=discrete_random_variable)*) -> scalar</signature>
  <signature>(symbol(type=continuous_random_variable)*) -> scalar</signature>
  <property> </property>
</MMLdefinition>
```

C.2.9.3 variance

```
<MMLdefinition>
  <Name> variance </Name>
  <description>
    This computes the second centered moment, also known as the variance.
    Given  k unspecified scalar arguments they are treated as equiprobable
    values of a random variable and the "variance" is
    computed as the second moment about the mean  U.
      variance( a1, a2, ...  an)  = E( (X - U)^2 ).
    If the first argument is a symbol X of type
    "discrete_random_variable", then all arguments are treated as
    discrete random variables, instead of data and the second moment
    about the mean is computed as  in section [7.7] (see reference below.)
      Sum( ( x_i - U )^2 * f(x_i) , x_i  in S )
    as
    where the probability that x = x_i is  P( x = x_i) = f(x_i) .
    The arguments are either all data, all discrete random variables,
    or all continuous random variables.
    The generalizes to continuous distributions and to
    k dimenions following the definitions found in:
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996, [7.1.2] and [7.7]
    </Reference>
  </description>
  <MMLattribute>
    <name>type</name>
    <values> random_variable | continuous_random_variable | data </value>
    <default> data </default>
  </MMLattribute>
  <functorclass>Nary , Operator </functorclass>
  <signature>(scalar*) -> scalar</signature>
  <signature>(scalar(type=data)*) -> scalar</signature>
  <signature>(symbol(type=discrete_random_variable)*) -> scalar</signature>
  <signature>(symbol(type=continuous_random_variable)*) -> scalar</signature>
</MMLdefinition>
```

C.2.9.4 median

```
<MMLdefinition>
  <Name> median </Name>
  <description>
    This represents the median of n data values.
    If n =2k + 1  then the mode is x_k.
    If n = 2k then the median is (x_k + x_(k+1)/2).
    (Note this discription assumes that the data has been
     sorted into ascending order.)
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996,  [7.7]
    </Reference>
  </description>
  <functorclass>Nary , Operator</functorclass>
  <signature>(scalar*) -> scalar</signature>
</MMLdefinition>
```

C.2.9.5 mode

```
<MMLdefinition>
  <Name> mode </Name>
  <description>
    This represents the mode of  n data values.
    The mode is the data value that occurs with the
    greatest frequency.
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996,  [7.7]
    </Reference>
  </description>
  <functorclass>Nary , Operator</functorclass>
  <signature>(scalar*) -> scalar</signature>
</MMLdefinition>
```

C.2.9.6 moment

```
<MMLdefinition>
  <Name> moment </Name>
  <description>
    This computes the ith moment of a set of data, or a random variable..
    Given  k  scalar arguments of unspecified type, they are treated
    as equiprobable  values of a random variable. and the "moments" are
    computed as the second moment about the mean  U.
      moment( degree=i, scalar*)= E( X^i ).
    If the first data argument x1 is a symbol X of type
    "discrete_random_variable", then all arguments are treated as
    discrete random variables, instead of data and the ith moment
    about the mean is computed as
      Sum( (x)^i * f(x) , x in S )
where the probability that x = x_i is P( x = x_i) = f(x_i) .
    The arguments are either all data, all discrete random variables,
    or all continuous random variables.
    The generalizes to continuous distributions and to
    k dimenions following the definitions found in:
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996, [7.1.2]
    </Reference>
  </description>
  <MMLattribute>
    <name>type</name>
    <values> random_variable | continuous_random_variable | data </value>
    <default> data </default>
  </MMLattribute>
  <functorclass>Nary , Operator </functorclass>
  <signature>(degree,scalar*) -> scalar</signature>
  <signature>(degree,scalar(type=data)*) -> scalar</signature>
  <signature>(degree,symbol(type=discrete_random_variable)*) -> scalar</signature>
  <signature>(degree, symbol(type=continuous_random_variable)*) -> scalar</signature>
</MMLdefinition>
```

C.2.10 Lineary Algebra

C.2.10.1 vector

```
<MMLdefinition>
  <Name> vector </Name>
  <description>
    A vector is an ordered n-tuple of values
    representing an element of an n-dimensional
    vector space.  The "values" are all from the
    same ring, typically real or complex.  They may
    be numbers, symbols, or general algebraic expressions.
    The type attribute can be used to specify the type of
    vector that is represented.
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996, [2.4]
    </Reference>
  </description>
  <MMLattribute>
    <name> type </name>
    <value>  real | complex | symbolic | anything </value>
    <default> real </default>
  </MMLattribute>
  <MMLattribute>
    <name> other </name>
    <value>  row | column </value>
    <default> row </default>
  </MMLattribute>
  <functorclass> constructor , N-ary </functorclass>
  <signature>
    ((cn|ci|apply)*) -> vector(type=real)
  </signature>
  <signature>
    [type=vectortype]((cn|ci|apply)*) -> vector(type=vectortype)
  </signature>
  <!-- Note that there is a notational need for expressing a sequence
       v1, v2, ... vn with an in-explicit value of n .  Also, in the
       following property, it should be clarified that b,v1, and v2 are all
       elements of the same ring. -->
  <property>  <!-- scalar multiplication-->
      <apply><forall/>
        <bvar><ci>b</ci></bvar>
        <bvar><ci>v1</ci></bvar>
        <bvar><ci>v2</ci></bvar>
        <reln>
          <apply><times/>
            <ci>ci>b</ci>
            <vector><ci>ci>v1</ci><ci>ci>v2</ci></vector>
            </apply>
            <vector>
              <apply><ci>b</ci><ci>v1</ci></apply>
```

```
            <apply><ci>b</ci><ci>v2</ci></apply>
          </vector>
      </reln>
    </apply>
  </property>
  <property> vector addition </property>
  <property> distributive over scalars</property>
  <property> associativity.</property>
  <property> Matrix * column vector </property>
  <property> row vector * Matrix </property>
  </property>
</MMLdefinition>
```

C.2.10.2 matrix

```
<MMLdefinition>
  <Name> matrix </Name>
  <description>
    This is the constructor for a matrix.  The matrix is
    constructed from matrix rows.  The type and properties
    spell out the normal interaction with vectors and
    scalars.
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996, [2.5.1]
    </Reference>
  </description>
  <MMLattribute>
    <name>type</name>
    <value>real | complex | integer | symbolic | anything </value>
    <default> real </default>
  </MMLattribute>
  <functorclass>constructor , N-ary </functorclass>
  <signature>(matrixrow*) -> matrix</signature>
  <signature>
    [type=matrixtype](matrixrow*) ->
       matrix(type=matrixtype)</signature>
  <property>scalar multiplication </property>
  <property>Matrix*column vector</property>
  <property>Addition</property>
  <property>Matrix*Matrix</property>
</MMLdefinition>
```

C.2.10.3 matrixrow

```
<MMLdefinition>
  <Name> matrixrow </Name>
  <description>
    This is a constructor for describing the rows of a matrix.
    This only occurs inside a matrix.  Its "type" is determined
    from the containing matrix element.
  </description>
  <functorclass>constructor , N-ary</functorclass>
  <signature>(cn|ci|apply)->matrixrow </signature>
</MMLdefinition>
```

C.2.10.4 determinant

```
<MMLdefinition>
  <Name>determinant</Name>
  <description>The "determinant" of a matrix.
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996, [2.5.4]
    </Reference>
  </description>
  <functorclass>Unary, operator</functorclass>
  <signature>(matrix)-> scalar </signature>
  </MMLdefinition>
```

C.2.10.5 transpose

```
<MMLdefinition>
  <Name> transpose </Name>
  <description>The transpose of a matrix or vector.
    <Reference> CRC Standard Mathematical Tables and Formulae,
      editor: Dan Zwillinger, CRC Press Inc., 1996, [2.4] and [2.5.1]
    </Reference>
  </description>
  <functorclass>Unary, Operator</functorclass>
  <signature>(vector)->vector(other=row)</signature>
  <signature>[other=column](vector)->vector(other=row)</signature>
  <signature>[other=row](vector)->vector(other=column)</signature>
  <signature>(matrix)->matrix</signature>
  <property>transpose(transpose(A))= A</property>
  <property>transpose(transpose(V))= V</property>
</MMLdefinition>
```

C.2.10.6 selector

```
<MMLdefinition>
  <Name> selector </Name>
  <description>
    The operator used to extract sub-objects from vectors, matrices
    matrix rows and lists.
    Elements are accessed by providing one index element for each
    dimension.  For Matrices, sub-matrices are selected by providing
    one fewer index items.  For a matrix A and a column vector V :
     select( i,j , A ) is the  i,j th element of A.
     select(i , A )  is the matrixrow formed from the ith row of A.
     select( i , V ) is the ith element of V.
     select( V ) is the sequence of all elements of V.
     select(A) is the sequence of all elements of A, extracted row
     by row.
     select(i,L) is the ith element of a list.
     select(L) is the sequence of elements of a list.
  </description>
  <functorclass>N-ary, operator)</functorclass>
  <signature>(scalar,scalar,matrix)->scalar</signature>
  <signature>(scalar,matrix)->matrixrow</signature>
  <signature>(matrix)->scalar* </property>
  <signature>(scalar,(vector|list|matrixrow))->scalar</signature>
  <signature>(vector|list|matrixrow)->scalar*</signature>
  <property>
    Forall(
       bvar(A(type=matrix)),bvar(V(type=vector)),
       select(A) = select(V)
     )
  </property>
  <property>For all vectors V, V = vector(select(V))</property>
</MMLdefinition>
```

Appendix D

Operator Dictionary (Non-Normative)

The following table gives the suggested dictionary of rendering properties for operators, fences, separators, and accents in MathML, all of which are represented by `mo` elements. For brevity, all such elements will be called simply 'operators' in this Appendix.

D.1 Format of operator dictionary entries

The operators are divided into groups, which are separated by blank lines in the listing below. The grouping, and the order of the groups, is significant for the proper grouping of sub-expressions using `<mrow>` (Section 3.3.1); the rule described there is especially relevant to the automatic generation of MathML by conversion from other formats for displayed mathematics, such as TEX, which do not always specify how sub-expressions nest.

The format of the table entries is: the `<mo>` element content between double quotes (start and end tags not shown), followed by the attribute list in XML format, starting with the `form` attribute, followed by the default rendering attributes which should be used for `mo` elements with the given content and `form` attribute.

Any attribute not listed for some entry has its default value, which is given in parentheses in the table of attributes in Section 3.2.4.

Note that the characters & and < are represented in the following table entries by the entity references `&` and `<` respectively, as would be necessary if they appeared in the content of an actual `mo` element (or any other MathML or XML element).

For example, the first entry,

```
"(" form="prefix" fence="true" stretchy="true" lspace="0em" rspace="0em"
```

could be expressed as an `mo` element by:

```
<mo form="prefix" fence="true" stretchy="true" lspace="0em" rspace="0em"> ( </mo>
```

(note the lack of double quotes around the content, and the whitespace added around the content for readability, which is optional in MathML).

This entry means that, for MathML renderers which use this suggested operator dictionary, giving the element `<mo form="prefix"> ( </mo>` alone, or simply `<mo> ( </mo>` in a position for which `form="prefix"` would be inferred (see below), is equivalent to giving the element with all attributes as shown above.

D.2 Indexing of operator dictionary

Note that the dictionary is indexed not just by the element content, but by the element content and `form` attribute value, together. Operators with more than one possible form have more than one entry. The MathML specification describes how the renderer chooses ('infers') which form to use when no `form` attribute is given; see Section 3.2.4.7.

Having made that choice, or with the `form` attribute explicitly specified in the `<mo>` element's start tag, the MathML renderer uses the remaining attributes from the dictionary entry for the appropriate single form of that operator, ignoring the entries for the other possible forms.

D.3 Choice of entity names

Extended characters in MathML (and in the operator dictionary below) are represented by XML-style entity references using the syntax `&character-name;` the complete list of characters and character names is given in Chapter 6. Many characters can be referred to by more than one name; often, memorable names composed of full words have been provided in MathML, as well as one or more names used in other standards, such as Unicode. The characters in the operators in this dictionary are generally listed under their full-word names when these exist. For example, the integral operator is named below by the one-character sequence `∫`, but could equally well be named `∫`. The choice of name for a given character in MathML has no effect on its rendering.

It is intended that every entity named below appears somewhere in Chapter 6. If this is not true, it is an error in this specification. If such an error exists, the abovementioned chapter should be taken as definitive, rather than this appendix.

D.4 Notes on `lspace` and `rspace` attributes

The values for `lspace` and `rspace` given here range from 0 to `verythickmathspace`, which has a default value of 6/18 em. For the invisible operators whose content is `⁢` or `⁡`, it is suggested that MathML renderers choose spacing in a context-sensitive way (which is an exception to the static values given in the following table). For `<mo>⁡</mo>`, the total spacing (`lspace+rspace`) in expressions such as 'sin x' (where the right operand doesn't start with a fence) should be greater than zero; for `<mo>⁢</mo>`, the total spacing should be greater than zero when both operands (or the nearest tokens on either side, if on the baseline) are identifiers displayed in a non-slanted font (i.e. under the suggested rules, when both operands are multi-character identifiers).

Some renderers may wish to use no spacing for most operators appearing in scripts (i.e. when `scriptlevel` is greater than 0; see Section 3.3.4), as is the case in TEX.

D.5 Operator dictionary entries

```
"("                             form="prefix"  fence="true" stretchy="true"  lspace="0em" rspace="0em"
")"                             form="postfix" fence="true" stretchy="true"  lspace="0em" rspace="0em"
"["                             form="prefix"  fence="true" stretchy="true"  lspace="0em" rspace="0em"
"]"                             form="postfix" fence="true" stretchy="true"  lspace="0em" rspace="0em"
"{"                             form="prefix"  fence="true" stretchy="true"  lspace="0em" rspace="0em"
"}"                             form="postfix" fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&CloseCurlyDoubleQuote;"       form="postfix" fence="true"  lspace="0em" rspace="0em"
"&CloseCurlyQuote;"             form="postfix" fence="true"  lspace="0em" rspace="0em"
"&LeftAngleBracket;"            form="prefix"  fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&LeftBracketingBar;"           form="prefix"  fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&LeftCeiling;"                 form="prefix"  fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&LeftDoubleBracket;"           form="prefix"  fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&LeftDoubleBracketingBar;"     form="prefix"  fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&LeftFloor;"                   form="prefix"  fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&OpenCurlyDoubleQuote;"        form="prefix"  fence="true"  lspace="0em" rspace="0em"
"&OpenCurlyQuote;"              form="prefix"  fence="true"  lspace="0em" rspace="0em"
"&RightAngleBracket;"           form="postfix" fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&RightBracketingBar;"          form="postfix" fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&RightCeiling;"                form="postfix" fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&RightDoubleBracket;"          form="postfix" fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&RightDoubleBracketingBar;"    form="postfix" fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&RightFloor;"                  form="postfix" fence="true" stretchy="true"  lspace="0em" rspace="0em"
"&LeftSkeleton;"                form="prefix"  fence="true"  lspace="0em" rspace="0em"
"&RightSkeleton;"               form="postfix" fence="true"  lspace="0em" rspace="0em"
"&InvisibleComma;"              form="infix"   separator="true"  lspace="0em" rspace="0em"
```

```
","                          form="infix"   separator="true"  lspace="0em" rspace="verythickmathspace"
"&HorizontalLine;"           form="infix"   stretchy="true" minsize="0"  lspace="0em" rspace="0em"
"&VerticalLine;"             form="infix"   stretchy="true" minsize="0"  lspace="0em" rspace="0em"
";"                          form="infix"   separator="true"  lspace="0em" rspace="thickmathspace"
";"                          form="postfix" separator="true"  lspace="0em" rspace="0em"
":="                         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Assign;"                   form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Because;"                  form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Therefore;"                form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&VerticalSeparator;"        form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"//"                         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Colon;"                    form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&"                      form="prefix"   lspace="0em" rspace="thickmathspace"
"&"                      form="postfix"  lspace="thickmathspace" rspace="0em"
"*="                         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"-="                         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"+="                         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"/="                         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"->"                         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
":"                          form="infix"    lspace="thickmathspace" rspace="thickmathspace"
".."                         form="postfix"  lspace="mediummathspace" rspace="0em"
"..."                        form="postfix"  lspace="mediummathspace" rspace="0em"
"&SuchThat;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&DoubleLeftTee;"            form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&DoubleRightTee;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&DownTee;"                  form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LeftTee;"                  form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&RightTee;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Implies;"                  form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&RoundImplies;"             form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"|"                          form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"||"                         form="infix"    lspace="mediummathspace" rspace="mediummathspace"
"&Or;"                       form="infix"   stretchy="true"  lspace="mediummathspace" rspace="mediummathspace"
"&&"                     form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&And;"                      form="infix"   stretchy="true"  lspace="mediummathspace" rspace="mediummathspace"
"&"                      form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"!"                          form="prefix"   lspace="0em" rspace="thickmathspace"
"&Not;"                      form="prefix"   lspace="0em" rspace="thickmathspace"
"&Exists;"                   form="prefix"   lspace="0em" rspace="thickmathspace"
"&ForAll;"                   form="prefix"   lspace="0em" rspace="thickmathspace"
"&NotExists;"                form="prefix"   lspace="0em" rspace="thickmathspace"
"&Element;"                  form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotElement;"               form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotReverseElement;"        form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSquareSubset;"          form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSquareSubsetEqual;"     form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSquareSuperset;"        form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSquareSupersetEqual;"   form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSubset;"                form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSubsetEqual;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSuperset;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
```

```
"&NotSupersetEqual;"          form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&ReverseElement;"            form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SquareSubset;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SquareSubsetEqual;"         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SquareSuperset;"            form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SquareSupersetEqual;"       form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Subset;"                    form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SubsetEqual;"               form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Superset;"                  form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SupersetEqual;"             form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&DoubleLeftArrow;"           form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&DoubleLeftRightArrow;"      form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&DoubleRightArrow;"          form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&DownLeftRightVector;"       form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&DownLeftTeeVector;"         form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&DownLeftVector;"            form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&DownLeftVectorBar;"         form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&DownRightTeeVector;"        form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&DownRightVector;"           form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&DownRightVectorBar;"        form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LeftArrow;"                 form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LeftArrowBar;"              form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LeftArrowRightArrow;"       form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LeftRightArrow;"            form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LeftRightVector;"           form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LeftTeeArrow;"              form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LeftTeeVector;"             form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LeftVector;"                form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LeftVectorBar;"             form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LowerLeftArrow;"            form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&LowerRightArrow;"           form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&RightArrow;"                form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&RightArrowBar;"             form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&RightArrowLeftArrow;"       form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&RightTeeArrow;"             form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&RightTeeVector;"            form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&RightVector;"               form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&RightVectorBar;"            form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&ShortLeftArrow;"            form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&ShortRightArrow;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&UpperLeftArrow;"            form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&UpperRightArrow;"           form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"="                           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"<"                        form="infix"    lspace="thickmathspace" rspace="thickmathspace"
">"                           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"!="                          form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"=="                          form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"<="                       form="infix"    lspace="thickmathspace" rspace="thickmathspace"
">="                          form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Congruent;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&CupCap;"                    form="infix"    lspace="thickmathspace" rspace="thickmathspace"
```

```
"&DotEqual;"                     form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&DoubleVerticalBar;"            form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Equal;"                        form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&EqualTilde;"                   form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Equilibrium;"                  form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&GreaterEqual;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&GreaterEqualLess;"             form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&GreaterFullEqual;"             form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&GreaterGreater;"               form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&GreaterLess;"                  form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&GreaterSlantEqual;"            form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&GreaterTilde;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&HumpDownHump;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&HumpEqual;"                    form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LeftTriangle;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LeftTriangleBar;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LeftTriangleEqual;"            form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&le;"                           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LessEqualGreater;"             form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LessFullEqual;"                form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LessGreater;"                  form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LessLess;"                     form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LessSlantEqual;"               form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&LessTilde;"                    form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NestedGreaterGreater;"         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NestedLessLess;"               form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotCongruent;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotCupCap;"                    form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotDoubleVerticalBar;"         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotEqual;"                     form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotEqualTilde;"                form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotGreater;"                   form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotGreaterEqual;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotGreaterFullEqual;"          form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotGreaterGreater;"            form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotGreaterLess;"               form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotGreaterSlantEqual;"         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotGreaterTilde;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotHumpDownHump;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotHumpEqual;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLeftTriangle;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLeftTriangleBar;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLeftTriangleEqual;"         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLess;"                      form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLessEqual;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLessFullEqual;"             form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLessGreater;"               form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLessLess;"                  form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLessSlantEqual;"            form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotLessTilde;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotNestedGreaterGreater;"      form="infix"    lspace="thickmathspace" rspace="thickmathspace"
```

```
"&NotNestedLessLess;"          form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotPrecedes;"                form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotPrecedesEqual;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotPrecedesSlantEqual;"      form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotPrecedesTilde;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotRightTriangle;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotRightTriangleBar;"        form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotRightTriangleEqual;"      form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSucceeds;"                form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSucceedsEqual;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSucceedsSlantEqual;"      form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotSucceedsTilde;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotTilde;"                   form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotTildeEqual;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotTildeFullEqual;"          form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotTildeTilde;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&NotVerticalBar;"             form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Precedes;"                   form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&PrecedesEqual;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&PrecedesSlantEqual;"         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&PrecedesTilde;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Proportion;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Proportional;"               form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&ReverseEquilibrium;"         form="infix"   stretchy="true"  lspace="thickmathspace" rspace="thickmathspace"
"&RightTriangle;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&RightTriangleBar;"           form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&RightTriangleEqual;"         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Succeeds;"                   form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SucceedsEqual;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SucceedsSlantEqual;"         form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SucceedsTilde;"              form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&Tilde;"                      form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&TildeEqual;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&TildeFullEqual;"             form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&TildeTilde;"                 form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&UpTee;"                      form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&VerticalBar;"                form="infix"    lspace="thickmathspace" rspace="thickmathspace"
"&SquareUnion;"                form="infix"   stretchy="true"  lspace="mediummathspace" rspace="mediummathspace"
"&Union;"                      form="infix"   stretchy="true"  lspace="mediummathspace" rspace="mediummathspace"
"&UnionPlus;"                  form="infix"   stretchy="true"  lspace="mediummathspace" rspace="mediummathspace"
"-"                            form="infix"    lspace="mediummathspace" rspace="mediummathspace"
"+"                            form="infix"    lspace="mediummathspace" rspace="mediummathspace"
"&Intersection;"               form="infix"   stretchy="true"  lspace="mediummathspace" rspace="mediummathspace"
"&MinusPlus;"                  form="infix"    lspace="mediummathspace" rspace="mediummathspace"
"&PlusMinus;"                  form="infix"    lspace="mediummathspace" rspace="mediummathspace"
"&SquareIntersection;"         form="infix"   stretchy="true"  lspace="mediummathspace" rspace="mediummathspace"
"&Vee;"                        form="prefix"  largeop="true" movablelimits="true" stretchy="true"  lspace="0em" rspace="thinmathspace"
"&CircleMinus;"                form="prefix"  largeop="true" movablelimits="true"  lspace="0em" rspace="thinmathspace"
"&CirclePlus;"                 form="prefix"  largeop="true" movablelimits="true"  lspace="0em" rspace="thinmathspace"
"&Sum;"                        form="prefix"  largeop="true" movablelimits="true" stretchy="true"  lspace="0em" rspace="thinmathspace"
"&Union;"                      form="prefix"  largeop="true" movablelimits="true" stretchy="true"  lspace="0em" rspace="thinmathspace"
```

```
"&UnionPlus;"                       form="prefix"  largeop="true" movablelimits="true" stretchy="true"  lspace="0em" rspace="thinmathspace"
"lim"                               form="prefix"  movablelimits="true"  lspace="0em" rspace="thinmathspace"
"max"                               form="prefix"  movablelimits="true"  lspace="0em" rspace="thinmathspace"
"min"                               form="prefix"  movablelimits="true"  lspace="0em" rspace="thinmathspace"
"&CircleMinus;"                     form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&CirclePlus;"                      form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&ClockwiseContourIntegral;"        form="prefix"  largeop="true" stretchy="true"  lspace="0em" rspace="0em"
"&ContourIntegral;"                 form="prefix"  largeop="true" stretchy="true"  lspace="0em" rspace="0em"
"&CounterClockwiseContourIntegral;" form="prefix"  largeop="true" stretchy="true"  lspace="0em" rspace="0em"
"&DoubleContourIntegral;"           form="prefix"  largeop="true" stretchy="true"  lspace="0em" rspace="0em"
"&Integral;"                        form="prefix"  largeop="true" stretchy="true"  lspace="0em" rspace="0em"
"&Cup;"                             form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&Cap;"                             form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&VerticalTilde;"                   form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&Wedge;"                           form="prefix"  largeop="true" movablelimits="true" stretchy="true"  lspace="0em" rspace="thinmathspace"
"&CircleTimes;"                     form="prefix"  largeop="true" movablelimits="true"  lspace="0em" rspace="thinmathspace"
"&Coproduct;"                       form="prefix"  largeop="true" movablelimits="true" stretchy="true"  lspace="0em" rspace="thinmathspace"
"&Product;"                         form="prefix"  largeop="true" movablelimits="true" stretchy="true"  lspace="0em" rspace="thinmathspace"
"&Intersection;"                    form="prefix"  largeop="true" movablelimits="true" stretchy="true"  lspace="0em" rspace="thinmathspace"
"&Coproduct;"                       form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&Star;"                            form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&CircleDot;"                       form="prefix"  largeop="true" movablelimits="true"  lspace="0em" rspace="thinmathspace"
"*"                                 form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&InvisibleTimes;"                  form="infix"    lspace="0em" rspace="0em"
"&CenterDot;"                       form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&CircleTimes;"                     form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&Vee;"                             form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&Wedge;"                           form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&Diamond;"                         form="infix"    lspace="thinmathspace" rspace="thinmathspace"
"&Backslash;"                       form="infix"   stretchy="true"  lspace="thinmathspace" rspace="thinmathspace"
"/"                                 form="infix"   stretchy="true"  lspace="thinmathspace" rspace="thinmathspace"
"-"                                 form="prefix"   lspace="0em" rspace="veryverythinmathspace"
"+"                                 form="prefix"   lspace="0em" rspace="veryverythinmathspace"
"&MinusPlus;"                       form="prefix"   lspace="0em" rspace="veryverythinmathspace"
"&PlusMinus;"                       form="prefix"   lspace="0em" rspace="veryverythinmathspace"
"."                                 form="infix"    lspace="0em" rspace="0em"
"&Cross;"                           form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"**"                                form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"&CircleDot;"                       form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"&SmallCircle;"                     form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"&Square;"                          form="prefix"   lspace="0em" rspace="verythinmathspace"
"&Del;"                             form="prefix"   lspace="0em" rspace="verythinmathspace"
"&PartialD;"                        form="prefix"   lspace="0em" rspace="verythinmathspace"
"&CapitalDifferentialD;"            form="prefix"   lspace="0em" rspace="verythinmathspace"
"&DifferentialD;"                   form="prefix"   lspace="0em" rspace="verythinmathspace"
"&Sqrt;"                            form="prefix"  stretchy="true"  lspace="0em" rspace="verythinmathspace"
"&DoubleDownArrow;"                 form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&DoubleLongLeftArrow;"             form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&DoubleLongLeftRightArrow;"        form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&DoubleLongRightArrow;"            form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&DoubleUpArrow;"                   form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
```

```
"&DoubleUpDownArrow;"            form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&DownArrow;"                    form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&DownArrowBar;"                 form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&DownArrowUpArrow;"             form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&DownTeeArrow;"                 form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LeftDownTeeVector;"            form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LeftDownVector;"               form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LeftDownVectorBar;"            form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LeftUpDownVector;"             form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LeftUpTeeVector;"              form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LeftUpVector;"                 form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LeftUpVectorBar;"              form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LongLeftArrow;"                form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LongLeftRightArrow;"           form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&LongRightArrow;"               form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&ReverseUpEquilibrium;"         form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&RightDownTeeVector;"           form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&RightDownVector;"              form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&RightDownVectorBar;"           form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&RightUpDownVector;"            form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&RightUpTeeVector;"             form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&RightUpVector;"                form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&RightUpVectorBar;"             form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&ShortDownArrow;"               form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"&ShortUpArrow;"                 form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"&UpArrow;"                      form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&UpArrowBar;"                   form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&UpArrowDownArrow;"             form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&UpDownArrow;"                  form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&UpEquilibrium;"                form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"&UpTeeArrow;"                   form="infix"   stretchy="true"  lspace="verythinmathspace" rspace="verythinmathspace"
"^"                              form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"<>"                           form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"’"                              form="postfix"  lspace="verythinmathspace" rspace="0em"
"!"                              form="postfix"  lspace="verythinmathspace" rspace="0em"
"!!"                             form="postfix"  lspace="verythinmathspace" rspace="0em"
"~"                              form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"@"                              form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"--"                             form="postfix"  lspace="verythinmathspace" rspace="0em"
"--"                             form="prefix"   lspace="0em" rspace="verythinmathspace"
"++"                             form="postfix"  lspace="verythinmathspace" rspace="0em"
"++"                             form="prefix"   lspace="0em" rspace="verythinmathspace"
"&ApplyFunction;"                form="infix"    lspace="0em" rspace="0em"
"?"                              form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"_"                              form="infix"    lspace="verythinmathspace" rspace="verythinmathspace"
"&Breve;"                        form="postfix" accent="true"  lspace="0em" rspace="0em"
"&Cedilla;"                      form="postfix" accent="true"  lspace="0em" rspace="0em"
"&DiacriticalGrave;"             form="postfix" accent="true"  lspace="0em" rspace="0em"
"&DiacriticalDot;"               form="postfix" accent="true"  lspace="0em" rspace="0em"
"&DiacriticalDoubleAcute;"       form="postfix" accent="true"  lspace="0em" rspace="0em"
"&DiacriticalLeftArrow;"         form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
```

```
"&DiacriticalLeftRightArrow;"       form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&DiacriticalLeftRightVector;"      form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&DiacriticalLeftVector;"           form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&DiacriticalAcute;"                form="postfix" accent="true"  lspace="0em" rspace="0em"
"&DiacriticalRightArrow;"           form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&DiacriticalRightVector;"          form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&DiacriticalTilde;"                form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&DoubleDot;"                       form="postfix" accent="true"  lspace="0em" rspace="0em"
"&DownBreve;"                       form="postfix" accent="true"  lspace="0em" rspace="0em"
"&Hacek;"                           form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&Hat;"                             form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&OverBar;"                         form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&OverBrace;"                       form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&OverBracket;"                     form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&OverParenthesis;"                 form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&TripleDot;"                       form="postfix" accent="true"  lspace="0em" rspace="0em"
"&UnderBar;"                        form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&UnderBrace;"                      form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&UnderBracket;"                    form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
"&UnderParenthesis;"                form="postfix" accent="true" stretchy="true"  lspace="0em" rspace="0em"
```

Appendix E

Document Object Model for MathML (Non-Normative)

The following sections describe the interfaces that have been defined in the Document Object Model for MathML. Please refer to Chapter 8 for more information.

E.1 IDL Interfaces

E.1.1 Miscellaneous Object Definitions

Interface MathMLDOMImplementation

Extends: `DOMImplementation`

This interface extends the `DOMImplementation` interface by adding a method to create a top level `MathMLmathElement`.

IDL Definition

```
interface MathMLDOMImplementation: DOMImplementation {
  MathMLmathElement createMathMLmathElement(in Document hostDocument, in Node parent);
};
```

Methods

createMathMLmathElement

Creates a `MathMLmathElement` to correspond to a MathML `math` element. The `MathMLmathElement` is empty, having no child elements or non-default attributes.

Parameters

`Document`	`hostDocument`	The Document object containing the `math` element.
`Node`	`parent`	The `Node` that is to be the parent node of the math element. This may be `null`.

Return value

`MathMLmathElement`	The newly created `MathMLmathElement`.

Exceptions

HIERARCHY_REQUEST_ERR Raised if a `math` element is not allowed in either `hostDocument` or the `parent` node.

Interface MathMLDocumentFragment

Extends: `MathMLElement`

This interface is provided as a specialization of the DocumentFragment interface. The child `Nodes` of this `MathMLElement` must be `MathMLElements` or `Text` nodes. As with the `DocumentFragment` object, inserting a `MathMLDocumentFragment` into a `MathMLElement` which can accept children has the effect of inserting each of the top-level child `Nodes` of the fragment rather than the fragment itself. Note that `MathMLDocumentFragments` are frequently used in the DOM as values of `readonly attributes`, encapsulating, for instance, various collections of child elements. When used in this way, these objects are always understood to be *live*, in the sense that they changes to the document are immediately reflected in them.

IDL Definition

```
interface MathMLDocumentFragment: MathMLElement {
};
```

E.1.2 Generic MathML Elements

Interface MathMLElement

Extends: `Element`

All MathML element interfaces derive from this object, which derives from the basic DOM interface Element.

Note: At some point it is expected that CSS support for mathematics will be available. At that point, the `style` attribute of a MathML element should be accessed through the ElementCSSInlineStyle interface, which is defined in the CSS DOM specification.

IDL Definition

```
interface MathMLElement: Element {
  attribute DOMString className;
  attribute DOMString style;
  attribute DOMString id;
};
```

Attributes

className **of type** `DOMString` The `class` attribute of the element. See the discussion elsewhere in this document and the HTML definition of the `class` attribute.

style **of type** `DOMString` A string identifying the element's `style` attribute.

id **of type** `DOMString` The element's identifier. See the discussion elsewhere in this document and the HTML definition of the `id` attribute.

Interface MathMLmathElement

Extends: `MathMLElement`

This interface represents the top-level MathML `math` element. It may be useful for interfacing between the Document Object Model objects encoding an enclosing document and the MathML DOM elements that are its children. It may also be used for some purposes as a MathML DOM surrogate for a `Document` object. For instance, MathML-specific factory methods could be placed here, as could methods for creating MathML-specific `Iterators` or `TreeWalkers`.

IDL Definition

```
interface MathMLmathElement: MathMLElement {
  readonly attribute MathMLDocumentFragment declarations;
  attribute DOMString macros;
  attribute DOMString display;
  MathMLdeclareElement insertDeclaration(in MathMLdeclareElement newDeclaration, in unsigned long index);
  MathMLdeclareElement setDeclaration(in MathMLdeclareElement newDeclaration, in unsigned long index);
  MathMLdeclareElement removeDeclaration(in unsigned long index);
  MathMLDocumentFragment createMathMLDocumentFragment();
  MathMLElement createMathMLElement(in DOMString tagName);
};
```

Attributes

declarations **of type** `MathMLDocumentFragment`**, readonly** Provides access to the `declare` elements which are children of this `math` element, in a `MathMLDocumentFragment`. All `Node`s of this fragment must be `MathMLdeclareElement`s.

macros **of type** `DOMString` Represents the `macros` attribute of the `math` element. See Section 7.1.

display **of type** `DOMString` Represents the `display` attribute of the `math` element. This value is either `block` or `inline`. See Section 7.1.

Methods

insertDeclaration

This method inserts `newDeclaration` before the current `index`-th child `declare` element of this `MathMLmathElement`. If `index` is 0, `newDeclaration` is appended as the last child `declare` element.

Parameters

`MathMLdeclareElement` `newDeclaration` A `MathMLdeclareElement` to be inserted as the `index`th child `declare` element.

`unsigned long` `index` A one-based index into the list of child `declare` elements of this element giving the position before which ne

Return value

`MathMLdeclareElement` The `MathMLdeclareElement` child of this `MathMLmathElement` representing `newDeclaration` in the DOM.

This method raises no exceptions.

setDeclaration

This method inserts `newDeclaration` as the `index`-th child declaration of this `MathMLmathElement`. If there is already an `index`-th `declare` child element, it is replaced by `newDeclaration`.

Parameters

`MathMLdeclareElement` `newDeclaration` A `MathMLdeclareElement` to be inserted as the `index`th child `declare` element.

`unsigned long` `index` A one-based index into the list of child `declare` elements of this element giving the position at which `newDe`

Return value

`MathMLdeclareElement` The `MathMLdeclareElement` being inserted.

This method raises no exceptions.

removeDeclaration

This method removes the `MathMLdeclareElement` representing the `index`-th `declare` child element of this `math` element, and returns it to the caller. Note that `index` is the position in the list of `declare` element children, as opposed to the position in the list of all child `Node`s.

Parameters

`unsigned long` `index` The one-based index of the `declare` element to be removed.

Return value

`MathMLdeclareElement` The `MathMLdeclareElement` being removed as a child `Node` of this element.

This method raises no exceptions.

createMathMLDocumentFragment

Creates a new empty `MathMLDocumentFragment` element.

Return value

`MathMLDocumentFragment` The `MathMLDocumentFragment` created.

This method raises no exceptions.

createMathMLElement

This method creates a `MathMLElement` to represent a MathML element of type `tagName`. The returned `MathMLElement` should be of the correct derived type to represent the element. In addition, if there are known attributes with default values, Attr nodes representing them are automatically created and attached to the element. `MathMLElements` representing required child elements are created as empty elements.

Parameters

`DOMString` `tagName` The case-sensitive name of the element type to instantiate.

Return value

`MathMLElement` The `MathMLElement` created.

Exceptions

INVALID_CHARACTER_ERR Raised if the specified name contains an illegal character.

Interface MathMLSemanticsElement

Extends: `MathMLElement`

This interface represents the `semantics` element in MathML.

IDL Definition

```
interface MathMLSemanticsElement: MathMLElement {
  attribute MathMLElement body;
  MathMLElement getAnnotation(in unsigned long index);
  MathMLElement insertAnnotation(in MathMLElement newAnnotation, in unsigned long index);
  MathMLElement setAnnotation(in MathMLElement newAnnotation, in unsigned long index);
};
```

Attributes

body of type `MathMLElement` This attribute represents the first child of the `semantics` element, i.e. the child giving the ‘primary’ content represented by the element.

Methods

getAnnotation

This method gives access to the `index`-th ‘alternate’ content associated with a `semantics` element.

Parameters

`unsigned long` `index` The one-based index of the annotation being retrieved.

Return value

`MathMLElement` The `MathMLAnnotationElement` or `MathMLXMLAnnotationElement` representing the `index`-th `annotation` or `xml-annotation` ch`mantics` element. Note that all child elements of a `semantics` element other than the first are required to be of one of these types.

This method raises no exceptions.

insertAnnotation

This method inserts `newAnnotation` before the current `index`-th 'alternate' content associated with a `semantics` element. If `index` is 0, `newAnnotation` is appended as the last `annotation` or `xml-annotation` child of this element.

Parameters

`MathMLElement`	`newAnnotation`	A `MathMLAnnotationElement` or `MathMLXMLAnnotationElement` representing the new `annotation` or `xml-ann`
`unsigned long`	`index`	The position in the list of `annotation` or `xml-annotation` children before which `newAnnotation` is to be inserted. T

Return value

`MathMLElement` The `MathMLAnnotationElement` or `MathMLXMLAnnotationElement` child of this element that represents the new annotation in the DOI

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newAnnotation` is not a `MathMLAnnotationElement` or `MathMLXMLAnnotationElement`.

INDEX_SIZE_ERR Raised if `index` is greater than the current number of `annotation` or `xml-annotation` children of this `semantics` element.

setAnnotation

This method allows setting or replacement of the `index`-th 'alternate' content associated with a `semantics` element. If there is already an `annotation` or `xml-annotation` element with this index, it is replaced by `newAnnotation`.

Parameters

`MathMLElement`	`newAnnotation`	A `MathMLAnnotationElement` or `MathMLXMLAnnotationElement` representing the new value of the `indexth` an child of this `semantics` element.
`unsigned long`	`index`	The position in the list of `annotation` or `xml-annotation` children of this `semantics` element that is to be occup annotation element is numbered 1.

Return value

`MathMLElement` The `MathMLAnnotationElement` or `MathMLXMLAnnotationElement` child of this element that represents the new annotation in the DOI

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newAnnotation` is not a `MathMLAnnotationElement` or `MathMLXMLAnnotationElement`.

INDEX_SIZE_ERR Raised if `index` is greater than one more than the current number of `annotation` or `xml-annotation` children of this `semantics` element.

Interface MathMLAnnotationElement

Extends: `MathMLElement`

This interface represents the `annotation` element of MathML.

IDL Definition

```
interface MathMLAnnotationElement: MathMLElement {
  attribute DOMString body;
  attribute DOMString encoding;
};
```

Attributes

body **of type** `DOMString` Provides access to the content of an `annotation` element.

encoding **of type** `DOMString` Provides access to the `encoding` attribute of an `annotation` element.

Interface MathMLXMLAnnotationElement

Extends: `MathMLElement`

This interface represents the `xml-annotation` element of MathML.

IDL Definition

```
interface MathMLXMLAnnotationElement: MathMLElement {
  readonly attribute NodeList contents;
  attribute DOMString encoding;
};
```

Attributes

contents **of type** `NodeList`**, readonly** Provides access to the content of an `annotation` element, represented by XML DOM `Nodes`.

encoding **of type** `DOMString` Provides access to the `encoding` attriubte of an `annotation` element.

E.1.3 Presentation Elements

Interface MathMLPresentationElement

Extends: `MathMLElement`

This interface is provided to serve as a base interface for various MathML Presentation interfaces. It contains no new attributes or methods at this time; however, it is felt that the distinction between Presentation and Content MathML entities should be indicated in the MathMLElement heirarchy. In particular, future versions of the MathML DOM may add functionality on this interface; it may also serve as an aid to implementors.

IDL Definition

```
interface MathMLPresentationElement: MathMLElement {
};
```

E.1.3.1 Leaf Presentation Element Interfaces

Interface MathMLCharacterElement

Extends: `MathMLPresentationElement`

This interface supports the `mchar` element Section 3.2.8.

IDL Definition

```
interface MathMLCharacterElement: MathMLPresentationElement {
  attribute DOMString name;
};
```

Attributes

name **of type** `DOMString` The name of a non-ASCII character, taken from Chapter 6.

Interface MathMLGlyphElement

Extends: `MathMLPresentationElement`

This interface supports the `mglyph` element Section 3.2.9.

IDL Definition

```
interface MathMLGlyphElement: MathMLPresentationElement {
  attribute DOMString alt;
  attribute DOMString fontfamily;
  attribute unsigned long index;
};
```

Attributes

alt **of type** `DOMString` A string giving an alternate name for the character. Represents the `mglyph`'s `alt` attribute.

fontfamily **of type** `DOMString` A string representing the font family.

index **of type** `unsigned long` An unsigned integer giving the glyph's position within the font.

Interface MathMLSpaceElement

Extends: `MathMLPresentationElement`

This interface extends the `MathMLPresentationElement` interface for the MathML *space* element `mspace`. Note that this is *not* derived from `MathMLPresentationToken`, despite the fact that `mspace` is classified as a token element, since it does not carry the attributes declared for `MathMLPresentationToken`.

IDL Definition

```
interface MathMLSpaceElement: MathMLPresentationElement {
  attribute DOMString width;
  attribute DOMString height;
  attribute DOMString depth;
};
```

Attributes

width **of type** `DOMString` A string of the form '*number h-unit*'; represents the `width` attribute for the `mspace` element, if specified.

height **of type** `DOMString` A string of the form '*number v-unit*'; represents the `height` attribute for the `mspace` element, if specified.

depth **of type** `DOMString` A string of the form '*number v-unit*'; represents the `depth` attribute for the `mspace` element, if specified.

E.1.3.2 Presentation Token Element Interfaces

Interfaces representing the MathML Presentation token elements that may have content are described here.

Interface MathMLPresentationToken

Extends: `MathMLPresentationElement`

This interface extends the `MathMLElement` interface to include access for attributes specific to text presentation. It serves as the base class for all MathML presentation token elements. Access to the body of the element is via the `nodeValue` attribute inherited from `Node`. Elements that expose only the core presentation token attributes are directly supported by this object. These elements are:

`mi` identifier element
`mn` number element
`mtext` text element

IDL Definition

```
interface MathMLPresentationToken: MathMLPresentationElement {
  attribute DOMString fontsize;
  attribute DOMString fontweight;
  attribute DOMString fontstyle;
  attribute DOMString fontfamily;
  attribute DOMString color;
  readonly attribute MathMLDocumentFragment contents;
};
```

Attributes

fontsize **of type** `DOMString` The font size attribute for the element, if specified.

fontweight **of type** `DOMString` The font weight attribute for the element, if specified.

fontstyle **of type** `DOMString` The font style attribute for the element, if specified.

fontfamily **of type** `DOMString` The font family attribute for the element, if specified.

color **of type** `DOMString` The color attribute for the element, if specified.

contents **of type** `MathMLDocumentFragment`**, readonly** Returns the child `Nodes` of the element. These should consist only of `Text` nodes and possibly `MathML-GlyphElements` or `MathMLCharacterElements`. Should behave the same as the base class's `Node::children` attribute; however, it is provided here for clarity.

Interface MathMLOperatorElement

Extends: `MathMLPresentationToken`

This interface extends the `MathMLPresentationToken` interface for the MathML *operator* element `mo`.

IDL Definition

```
interface MathMLOperatorElement: MathMLPresentationToken {
  attribute DOMString form;
  attribute DOMString fence;
  attribute DOMString separator;
  attribute DOMString lspace;
  attribute DOMString rspace;
  attribute DOMString stretchy;
  attribute DOMString symmetric;
  attribute DOMString maxsize;
  attribute DOMString minsize;
  attribute DOMString largeop;
  attribute DOMString moveablelimits;
  attribute DOMString accent;
};
```

Attributes

form **of type** DOMString The form attribute (prefix, infix or postfix) for the mo element, if specified.
fence **of type** DOMString The fence attribute (true or false) for the mo element, if specified.
separator **of type** DOMString The separator attribute (true or false) for the mo element, if specified.
lspace **of type** DOMString The lspace attribute (spacing to left) of the mo element, if specified.
rspace **of type** DOMString The rspace attribute (spacing to right) of the mo element, if specified.
stretchy **of type** DOMString The stretchy attribute (true or false) for the mo element, if specified.
symmetric **of type** DOMString The symmetric attribute (true or false) for the mo element, if specified.
maxsize **of type** DOMString The maxsize attribute for the mo element, if specified.
minsize **of type** DOMString The minsize attribute for the mo element, if specified.
largeop **of type** DOMString The largeop attribute for the mo element, if specified.
moveablelimits **of type** DOMString The moveablelimits (true or false) attribute for the mo element, if specified.
accent **of type** DOMString The accent attribute (true or false) for the mo element, if specified.

Interface MathMLStringLitElement

Extends: MathMLPresentationToken

This interface extends the MathMLPresentationToken interface for the MathML *string literal* element ms.

IDL Definition

```
interface MathMLStringLitElement: MathMLPresentationToken {
  attribute DOMString lquote;
  attribute DOMString rquote;
};
```

Attributes

lquote **of type** DOMString A string giving the opening delimiter for the string literal; represents the lquote attribute for the ms element, if specified.
rquote **of type** DOMString A string giving the closing delimiter for the string literal; represents the rquote attribute for the ms element, if specified.

E.1.3.3 Presentation Container Interfaces

We include under the heading of Presentation Container Elements interfaces designed to represent MathML Presentation elements that can contain arbitrary numbers of child MathMLElements.

Interface MathMLPresentationContainer

Extends: MathMLPresentationElement

This interface represents MathML Presentation elements that may contain arbitarily many child elements.

IDL Definition

```
interface MathMLPresentationContainer: MathMLPresentationElement {
  readonly attribute MathMLDocumentFragment arguments;
  MathMLElement getArgument(in unsigned long index);
  MathMLElement insertArgument(in MathMLElement newArgument, in unsigned long index);
  MathMLElement setArgument(in MathMLElement newArgument, in unsigned long index);
  void deleteArgument(in unsigned long index);
  MathMLElement removeArgument(in unsigned long index);
};
```

Attributes

arguments of type `MathMLDocumentFragment`, readonly This attribute accesses the child `MathMLElements` of this element, as a `MathMLDocumentFragment`.

Methods

getArgument

This method returns the `index`-th child element of this element.

Parameters

`unsigned long` `index` The one-based index of the argument to be retrieved.

Return value

`MathMLElement` A `MathMLElement` representing the element being retrieved.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of child elements.

insertArgument

This method inserts `newArgument` before the current `index`-th child argument of this element. If `index` is 0, `newArgument` is appended as the last argument.

Parameters

`MathMLElement` `newArgument` A `MathMLElement` representing the element that is to be inserted as a child argument of this element.

`unsigned long` `index` The index of the position in the list of arguments before which `newArgument` is to be inserted. The first argument is numbe

Return value

`MathMLElement` The `MathMLElement` child of this element that represents the new argument in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newArgument` represents an element that cannot be an argument of this element.

INDEX_SIZE_ERR Raised if `index` is greater than the number of child arguments.

setArgument

This method sets `newArgument` as the `index`-th child element of this element. If there is already an element at position `index`, it is replaced by `newArgument`.

Parameters

`MathMLElement` `newArgument` A `MathMLElement` representing the element that is to be set as the `index`-th argument of this element.

`unsigned long` `index` The index of the argument that is to be set to `newArgument`.

Return value

`MathMLElement` The `MathMLElement` child of this element that represents the new argument in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newArgument` represents an element that cannot be an argument of this element.

INDEX_SIZE_ERR Raised if `index` is greater than one more than the number of child elements.

deleteArgument

This method deletes the `index`-th child `MathMLElement` of this element.

Parameters

`unsigned long` `index` The one-based index of the argument to be deleted.

Return value

`void` None.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of child elements.

removeArgument

This method removes the `index`th child element of this element and returns it to the caller.

Parameters

`unsigned long` `index` The one-based index of the argument to be removed.

Return value

`MathMLElement` A `MathMLElement` representing the child element being removed.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of child elements.

Interface MathMLStyleElement

Extends: `MathMLPresentationContainer`

This interface extends the `MathMLElement` interface for the MathML *style* element `mstyle`. While the `mstyle` element may contain any *attributes* allowable on any MathML presentation element, only *attributes* specific to the `mstyle` element are included in the interface below. Other attributes should be accessed using the methods on the base `Element` class, particularly the `Element::getAttribute` and `Element::setAttribute` methods, or even the `Node::attributes` attribute to access all of them at once. Not only does this obviate a lengthy list below, but it seems likely that most implementations will find this a considerably more useful interface to a `MathMLStyleElement`.

IDL Definition

```
interface MathMLStyleElement: MathMLPresentationContainer {
  attribute DOMString scriptlevel;
  attribute DOMString displaystyle;
  attribute DOMString scriptsizemultiplier;
  attribute DOMString scriptminsize;
  attribute DOMString color;
  attribute DOMString background;
};
```

Attributes

scriptlevel **of type** `DOMString` A string of the form '+/- *unsigned integer*'; represents the `scriptlevel` attribute for the `mstyle` element, if specified. See also the discussion of this attribute.

displaystyle **of type** `DOMString` Either `true` or `false`; a string representing the `displaystyle` attribute for the `mstyle` element, if specified. See also the discussion of this attribute.

scriptsizemultiplier **of type** `DOMString` A string of the form '*number*'; represents the `scriptsizemultiplier` attribute for the `mstyle` element, if specified. See also the discussion of this attribute.

scriptminsize **of type** `DOMString` A string of the form '*number v-unit*'; represents the `scriptminsize` attribute for the `mstyle` element, if specified. See also the discussion of this attribute.

color **of type** DOMString A string representation of a color; represents the `color` attribute for the `mstyle` element, if specified. See also the discussion of this attribute.

background **of type** DOMString A string representation of a color or the string `transparent`; represents the `background` attribute for the `mstyle` element, if specified. See also the discussion of this attribute.

Interface MathMLPaddedElement

Extends: MathMLPresentationContainer

This interface extends the `MathMLElement` interface for the MathML *spacing adjustment* element `mpadded`.

IDL Definition

```
interface MathMLPaddedElement: MathMLPresentationContainer {
  attribute DOMString width;
  attribute DOMString lspace;
  attribute DOMString height;
  attribute DOMString depth;
};
```

Attributes

width **of type** DOMString A string representing the total `width` of the `mpadded` element, if specified. See also the discussion of this attribute.

lspace **of type** DOMString A string representing the `lspace` attribute - the additional space to the left - of the `mpadded` element, if specified. See also the discussion of this attribute.

height **of type** DOMString A string representing the `height` above the baseline of the `mpadded` element, if specified. See also the discussion of this attribute.

depth **of type** DOMString A string representing the `depth` beneath the baseline of the `mpadded` element, if specified. See also the discussion of this attribute.

Interface MathMLFencedElement

Extends: MathMLPresentationContainer

This interface extends the `MathMLPresentationContainer` interface for the MathML *fenced content* element `mfenced`.

IDL Definition

```
interface MathMLFencedElement: MathMLPresentationContainer {
  attribute DOMString open;
  attribute DOMString close;
  attribute DOMString separators;
};
```

Attributes

open **of type** DOMString A string representing the *opening-fence* for the `mfenced` element, if specified; this is the element's `open` attribute.

close **of type** DOMString A string representing the *opening-fence* for the `mfenced` element, if specified; this is the element's `close` attribute.

separators **of type** DOMString A string representing any separating characters inside the `mfenced` element, if specified; this is the element's `separators` attribute.

Interface MathMLEncloseElement

Extends: `MathMLPresentationContainer`

This interface supports the `menclose` element Section 3.3.9.

IDL Definition

```
interface MathMLEncloseElement: MathMLPresentationContainer {
  attribute DOMString notation;
};
```

Attributes

notation **of type** `DOMString` A string giving a name for the notation enclosing the element's contents. Represents the `notation` attribute of the `menclose`. Allowed values are `longdiv`, `actuarial`, `radical`.

Interface MathMLActionElement

Extends: `MathMLPresentationContainer`

This interface extends the `MathMLPresentationContainer` interface for the MathML *enlivening expression* element `maction`.

IDL Definition

```
interface MathMLActionElement: MathMLPresentationContainer {
  attribute DOMString actiontype;
  attribute DOMString selection;
};
```

Attributes

actiontype **of type** `DOMString` A string specifying the action. Possible values include `toggle`, `statusline`, `tooltip`, `highlight`, and `menu`.

selection **of type** `DOMString` A string specifying an integer that selects the current subject of the action.

E.1.3.4 Presentation Schemata Interfaces

Interface MathMLFractionElement

Extends: `MathMLPresentationElement`

This interface extends the `MathMLPresentationElement` interface for the MathML *fraction* element `mfrac`.

IDL Definition

```
interface MathMLFractionElement: MathMLPresentationElement {
  attribute DOMString linethickness;
  attribute MathMLElement numerator;
  attribute MathMLElement denominator;
};
```

Attributes

linethickness **of type** `DOMString` A string representing the `linethickness` attribute of the `mfrac`, if specified.

numerator **of type** `MathMLElement` The first child `MathMLElement` of the `MathMLFractionElement`; represents the numerator of the represented fraction.

denominator **of type** `MathMLElement` The second child `MathMLElement` of the `MathMLFractionElement`; represents the denominator of the represented fraction.

Interface MathMLRadicalElement

Extends: `MathMLPresentationElement`

This interface extends the `MathMLPresentationElement` interface for the MathML *radical* and *square root* elements `mroot` and `msqrt`.

IDL Definition

```
interface MathMLRadicalElement: MathMLPresentationElement {
  attribute MathMLElement radicand;
  attribute MathMLElement index;
};
```

Attributes

radicand **of type** `MathMLElement` The first child `MathMLElement` of the `MathMLRadicalElement`; represents the *base* of the represented radical.

index **of type** `MathMLElement` The second child `MathMLElement` of the `MathMLRadicalElement`; represents the *index* of the represented radical. This must be `null` for `msqrt` elements.

Interface MathMLScriptElement

Extends: `MathMLPresentationElement`

This interface extends the `MathMLPresentationElement` interface for the MathML *subscript*, *superscript* and *subscript-superscript pair* elements `msub`, `msup`, and `msubsup`.

IDL Definition

```
interface MathMLScriptElement: MathMLPresentationElement {
  attribute DOMString subscriptshift;
  attribute DOMString superscriptshift;
  attribute MathMLElement base;
  attribute MathMLElement subscript;
  attribute MathMLElement superscript;
};
```

Attributes

subscriptshift **of type** `DOMString` A string representing the minimum amount to shift the baseline of the *subscript* down, if specified; this is the element's `subscriptshift` attribute. This must return `null` for an `msup`.

superscriptshift **of type** `DOMString` A string representing the minimum amount to shift the baseline of the *superscript* up, if specified; this is the element's `superscriptshift` attribute. This must return `null` for a `msub`.

base **of type** MathMLElement A `MathMLElement` representing the *base* of the script. This is the first child of the element.

subscript **of type** MathMLElement A `MathMLElement` representing the *subscript* of the script. This is the second child of a `msub` or `msubsup`; retrieval must return `null` for an `msup`.

HIERARCHY_REQUEST_ERR Raised when the element is a `msup`.

superscript **of type** MathMLElement A `MathMLElement` representing the *superscript* of the script. This is the second child of a `msup` or the third child of a `msubsup`; retrieval must return `null` for an `msub`.

HIERARCHY_REQUEST_ERR Raised when the element is a `msub`.

Interface MathMLUnderOverElement

Extends: MathMLPresentationElement

This interface extends the `MathMLPresentationElement` interface for the MathML *underscript*, *overscript* and *overscript-underscript pair* elements `munder`, `mover` and `munderover`.

IDL Definition

```
interface MathMLUnderOverElement: MathMLPresentationElement {
  attribute DOMString accentunder;
  attribute DOMString accent;
  attribute MathMLElement base;
  attribute MathMLElement underscript;
  attribute MathMLElement overscript;
};
```

Attributes

accentunder **of type** DOMString Either `true` or `false` if present; a string controlling whether *underscript* is drawn as an ‘accent’ or as a ‘limit’, if specified; this is the element’s `accentunder` attribute. This must return `null` for an `mover`.

accent **of type** DOMString Either `true` or `false` if present; a string controlling whether *overscript* is drawn as an ‘accent’ or as a ‘limit’, if specified; this is the element’s `accent` attribute. This must return `null` for an `munder`.

base **of type** MathMLElement A `MathMLElement` representing the *base* of the script. This is the first child of the element.

underscript **of type** MathMLElement A `MathMLElement` representing the *underscript* of the script. This is the second child of a `munder` or `munderover`; retrieval must return `null` for an `mover`.

HIERARCHY_REQUEST_ERR Raised when the element is a `mover`.

overscript **of type** MathMLElement A `MathMLElement` representing the *overscript* of the script. This is the second child of a `mover` or the third child of a `munderover`; retrieval must return `null` for an `munder`.

HIERARCHY_REQUEST_ERR Raised when the element is a `munder`.

Interface MathMLMultiScriptsElement

Extends: MathMLPresentationElement

This interface extends the `MathMLPresentationElement` interface for the MathML *multiscripts* (including prescripts or tensors) element `mmultiscripts`.

IDL Definition

```
interface MathMLMultiScriptsElement: MathMLPresentationElement {
  attribute DOMString subscriptshift;
  attribute DOMString superscriptshift;
  attribute MathMLElement base;
  readonly attribute MathMLDocumentFragment prescripts;
  readonly attribute MathMLDocumentFragment scripts;
  readonly attribute unsigned long numprescriptcolumns;
  readonly attribute unsigned long numscriptcolumns;
  MathMLElement getPreSubScript(in unsigned long colIndex);
  MathMLElement getSubScript(in unsigned long colIndex);
  MathMLElement getPreSuperScript(in unsigned long colIndex);
  MathMLElement getSuperScript(in unsigned long colIndex);
  MathMLElement insertPreSubScriptBefore(in unsigned long colIndex, in MathMLElement newScript);
  MathMLElement setPreSubScriptAt(in unsigned long colIndex, in MathMLElement newScript);
  MathMLElement insertSubScriptBefore(in unsigned long colIndex, in MathMLElement newScript);
  MathMLElement setSubScriptAt(in unsigned long colIndex, in MathMLElement newScript);
  MathMLElement insertPreSuperScriptBefore(in unsigned long colIndex, in MathMLElement newScript);
  MathMLElement setPreSuperScriptAt(in unsigned long colIndex, in MathMLElement newScript);
  MathMLElement insertSuperScriptBefore(in unsigned long colIndex, in MathMLElement newScript);
  MathMLElement setSuperScriptAt(in unsigned long colIndex, in MathMLElement newScript);
};
```

Attributes

subscriptshift **of type** `DOMString` A string representing the minimum amount to shift the baseline of the *subscripts* down, if specified; this is the element's `subscriptshift` attribute.

superscriptshift **of type** `DOMString` A string representing the minimum amount to shift the baseline of the *superscripts* up, if specified; this is the element's `superscriptshift` attribute.

base **of type** `MathMLElement` A `MathMLElement` representing the *base* of the script. This is the first child of the element.

prescripts **of type** `MathMLDocumentFragment`**, readonly** A `NodeList` representing the *prescripts* of the script, which appear in the order described by the expression *(prescript presuperscript)**. This is the same as traversing the contents of the `NodeList` returned by `Node::childNodes()` from the `Node` following the `<mprescripts/>` (if present) to the end of the list.

scripts **of type** `MathMLDocumentFragment`**, readonly** A `MathMLDocumentFragment` representing the *scripts* of the script, which appear in the order described by the expression *(script superscript)**. This is the same as traversing the contents of the `NodeList` returned by `Node::childNodes()` from the first `Node` up to and including the `Node` preceding the `<mprescripts/>` (if present).

numprescriptcolumns **of type** `unsigned long`**, readonly** The number of script/subscript columns preceding (to the left of) the *base*. Should always be half of `getprescripts().length()`

numscriptcolumns **of type** `unsigned long`**, readonly** The number of script/subscript columns following (to the right of) the *base*. Should always be half of `getscripts().length()`

Methods

getPreSubScript

A convenience method to retrieve *pre-subscript* children of the element, referenced by column index .

Parameters

`unsigned long`	`colIndex`	Column index of *prescript* (where 1 represents the leftmost *prescript* column).

Return value

`MathMLElement` Returns the `MathMLElement` representing the *colIndex*-th presubscript (to the left of the *base*, counting from 1 at the far left). Note that th `MathMLElement` corresponding to the special element `<none/>` in the case of a 'missing' *presubscript* (see the discussion of `mmultiscrip` be `null` if `colIndex` is out of range for the element.

This method raises no exceptions.

getSubScript

A convenience method to retrieve *subscript* children of the element, referenced by column index.

Parameters

`unsigned long` `colIndex` Column index of *script* (where 1 represents the leftmost *script* column, the first to the right of the *base*).

Return value

`MathMLElement` Returns the `MathMLElement` representing the *colIndex*-th subscript to the right of the *base*. Note that this may be the `MathMLElement` co the special element `<none/>` in the case of a 'missing' *subscript* (see the discussion of `mmultiscripts`), or it may be `null` if `colIndex` i for the element.

This method raises no exceptions.

getPreSuperScript

A convenience method to retrieve *pre-superscript* children of the element, referenced by column index .

Parameters

`unsigned long` `colIndex` Column index of *pre-superscript* (where 1 represents the leftmost *prescript* column).

Return value

`MathMLElement` Returns the `MathMLElement` representing the *colIndex*-th presuperscript (to the left of the *base*, counting from 1 at the far left). Note that th `MathMLElement` corresponding to the special element `<none/>` in the case of a 'missing' *presuperscript* (see the discussion of `mmultis` may be `null` if `colIndex` is out of range for the element.

This method raises no exceptions.

getSuperScript

A convenience method to retrieve *superscript* children of the element, referenced by column index .

Parameters

`unsigned long` `colIndex` Column index of *script* (where 1 represents the leftmost *script* column, the first to the right of the *base*)

Return value

`MathMLElement` Returns the `MathMLElement` representing the *colIndex*-th superscript to the right of the *base*. Note that this may be the `MathMLElement` c to the special element `<none/>` in the case of a 'missing' *superscript* (see the discussion of `mmultiscripts`), or it may be `null` if `colI` range for the element.

This method raises no exceptions.

insertPreSubScriptBefore

A convenience method to insert a *pre-subscript* before the position referenced by column index. If `colIndex` is 0, the new *pre-subscript* is appended as the last *pre-subscript* of the `mmultiscripts` element; if colIndex is 1, a new *pre-subscript* is prepended at the far left. Note that inserting a new *pre-subscript* will cause the insertion of an empty *pre-superscript* in the same column.

Parameters

`unsigned long` `colIndex` Column index of *pre-subscript* (where 1 represents the leftmost *prescript* column).

`MathMLElement` `newScript` A `MathMLElement` representing the element to be inserted as a *pre-subscript*.

Return value

`MathMLElement` The `MathMLElement` child of this `MathMLMultiScriptsElement` representing the new script in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newScript` represents an element that cannot be a pre-subscript.

INDEX_SIZE_ERR Raised if `colIndex` is greater than the number of *pre-scripts* of the element.

setPreSubScriptAt

A convenience method to set the *pre-subscript* child at the position referenced by `colIndex`. If there is currently a *pre-subscript* at this position, it is replaced by `newScript`.

Parameters

unsigned long	colIndex	Column index of *pre-subscript* (where 1 represents the leftmost *prescript* column).
MathMLElement	newScript	MathMLElement representing the element that is to be set as the colIndexth *pre-subscript* child of this element.

Return value

MathMLElement The MathMLElement child of this MathMLMultiScriptsElement representing the new pre-subscript in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if newScript represents an element that cannot be a *pre-subscript*.

INDEX_SIZE_ERR Raised if colIndex is greater than one more than the number of *pre-scripts* of the element.

insertSubScriptBefore

A convenience method to insert a *subscript* before the position referenced by column index. If colIndex is 0, the new *subscript* is appended as the last *subscript* of the mmultiscripts element; if colIndex is 1, a new *subscript* is prepended at the far left. Note that inserting a new *subscript* will cause the insertion of an empty *superscript* in the same column.

Parameters

unsigned long	colIndex	Column index of *subscript*, where 1 represents the leftmost *script* column (the first to the right of the *base*).
MathMLElement	newScript	A MathMLElement representing the element to be inserted as a *subscript*.

Return value

MathMLElement The MathMLElement child of this MathMLMultiScriptsElement that represents the new *subscript* in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if newScript represents an element that cannot be a subscript.

INDEX_SIZE_ERR Raised if colIndex is greater than the number of *scripts* of the element.

setSubScriptAt

A convenience method to set the *subscript* child at the position referenced by colIndex. If there is currently a *subscript* at this position, it is replaced by newScript.

Parameters

unsigned long	colIndex	Column index of *subscript*, where 1 represents the leftmost *script* column (the first to the right of the *base*).
MathMLElement	newScript	MathMLElement representing the element that is to be set as the colIndexth *subscript* child of this element.

Return value

MathMLElement The MathMLElement chlid of this element representing the new *subscript* in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if newScript represents an element that cannot be a *subscript*.

INDEX_SIZE_ERR Raised if colIndex is greater than one more than the number of *scripts* of the element.

insertPreSuperScriptBefore

A convenience method to insert a *pre-superscript* before the position referenced by column index. If colIndex is 0, the new *pre-superscript* is appended as the last *pre-superscript* of the mmultiscripts element; if colIndex is 1, a new *pre-superscript* is prepended at the far left. Note that inserting a new *pre-superscript* will cause the insertion of an empty *pre-subscript* in the same column.

Parameters

unsigned long	colIndex	Column index of *pre-superscript* (where 1 represents the leftmost *prescript* column).
MathMLElement	newScript	A MathMLElement representing the element to be inserted as a *pre-superscript*.

Return value

MathMLElement The MathMLElement child of this element that represents the new *pre-superscript* in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if newScript represents an element that cannot be a pre-superscript.

INDEX_SIZE_ERR Raised if colIndex is greater than the number of *pre-scripts* of the element.

setPreSuperScriptAt

A convenience method to set the *pre-superscript* child at the position referenced by colIndex. If there is currently a *pre-superscript* at this position, it is replaced by newScript.

Parameters

unsigned long	colIndex	Column index of *pre-superscript* (where 1 represents the leftmost *prescript* column).
MathMLElement	newScript	MathMLElement representing the element that is to be set as the colIndexth *pre-superscript* child of this element.

Return value

`MathMLElement`	The `MathMLElement` child of this element that represents the new *pre-superscript* in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newScript` represents an element that cannot be a *pre-superscript*.

INDEX_SIZE_ERR Raised if `colIndex` is greater than one more than the number of *pre-scripts* of the element.

insertSuperScriptBefore

A convenience method to insert a *superscript* before the position referenced by column index. If `colIndex` is 0, the new *superscript* is appended as the last *superscript* of the `mmultiscripts` element; if colIndex is 1, a new *superscript* is prepended at the far left. Note that inserting a new *superscript* will cause the insertion of an empty *subscript* in the same column.

Parameters

`unsigned long`	`colIndex`	Column index of *superscript*, where 1 represents the leftmost *script* column (the first to the right of the *base*).
`MathMLElement`	`newScript`	A `MathMLElement` representing the element to be inserted as a *superscript*.

Return value

`MathMLElement`	The `MathMLElement` chld of this element that represents the new *superscript* in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newScript` represents an element that cannot be a superscript.

INDEX_SIZE_ERR Raised if `colIndex` is greater than the number of *scripts* of the element.

setSuperScriptAt

A convenience method to set the *superscript* child at the position referenced by `colIndex`. If there is currently a *superscript* at this position, it is replaced by `newScript`.

Parameters

`unsigned long`	`colIndex`	Column index of *superscript*, where 1 represents the leftmost *script* column (the first to the right of the *base*).
`MathMLElement`	`newScript`	`MathMLElement` representing the element that is to be set as the `colIndex`th *superscript* child of this element.

Return value

`MathMLElement`	The `MathMLElement` child of this element that represents the new *superscript* in the DOM.

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newScript` represents an element that cannot be a *superscript*.

INDEX_SIZE_ERR Raised if `colIndex` is greater than one more than the number of *scripts* of the element.

Interface MathMLTableElement

Extends: `MathMLPresentationElement`

This interface extends the `MathMLPresentationElement` interface for the MathML *table or matrix* element `mtable`.

IDL Definition

```
interface MathMLTableElement: MathMLPresentationElement {
  attribute DOMString align;
  attribute DOMString rowalign;
  attribute DOMString columnalign;
  attribute DOMString groupalign;
  attribute DOMString alignmentscope;
  attribute DOMString columnwidth;
  attribute DOMString width;
  attribute DOMString rowspacing;
  attribute DOMString columnspacing;
  attribute DOMString rowlines;
  attribute DOMString columnlines;
  attribute DOMString frame;
  attribute DOMString framespacing;
  attribute DOMString equalrows;
  attribute DOMString equalcolumns;
  attribute DOMString displaystyle;
  attribute DOMString side;
  attribute DOMString minlabelspacing;
  readonly attribute MathMLDocumentFragment rows;
  MathMLTableRowElement insertEmptyRow(in long index);
  MathMLLabeledRowElement insertEmptyLabeledRow(in long index);
  MathMLTableRowElement getRow(in unsigned long index);
  MathMLTableRowElement insertRow(in long index, in MathMLTableRowElement newRow);
  MathMLTableRowElement setRow(in long index, in MathMLTableRowElement newRow);
  void deleteRow(in unsigned long index);
  MathMLTableRowElement removeRow(in long index);
};
```

Attributes

align **of type** `DOMString` A string representing the vertical alignment of the table with the adjacent text. Allowed values are (`top` | `bottom` | `center` | `baseline` | `axis`)[*rownumber*], where *rownumber* is between 1 and *n* (for a table with *n* rows) or -1 and -*n*.

rowalign **of type** `DOMString` A string representing the alignment of entries in each row, consisting of a space-separated sequence of alignment specifiers, each of which can have the following values: `top`, `bottom`, `center`, `baseline`, or `axis`.

columnalign **of type** `DOMString` A string representing the alignment of entries in each column, consisting of a space-separated sequence of alignment specifiers, each of which can have the following values: `left`, `center`, or `right`.

groupalign **of type** `DOMString` A string specifying how the alignment groups within the cells of each row are to be aligned with the corresponding items above or below them in the same column. The string consists of a sequence of braced group alignment lists. Each group alignment list is a space-separated sequence, each of which can have the following values: `left`, `right`, `center`, or `decimalpoint`.

alignmentscope **of type** `DOMString` A string consisting of the values `true` or `false` indicating, for each column, whether it can be used as an alignment scope.

columnwidth **of type** `DOMString` A string consisting of a space-separated sequence of specifiers, each of which can have one of the following forms: `auto`, `number h-unit`, `namedspace`, or `fit`. (A value of the form `namedspace` is one of `veryverythinmathspace`, `verythinmathspace`, `thinmathspace`, `mediummathspace`, `thickmathspace`, `verythickmathspace`, or `veryverythickmathspace`. This represents the element's `columnwidth` attribute.

width **of type** `DOMString` A string that is either of the form `number h-unit` or is the string `auto`. This represents the element's `width` attribute.

rowspacing **of type** DOMString A string consisting of a space-separated sequence of specifiers of the form `number v-unit` representing the space to be added between rows.

columnspacing **of type** DOMString A string consisting of a space-separated sequence of specifiers of the form `number h-unit` representing the space to be added between columns.

rowlines **of type** DOMString A string specifying whether and what kind of lines should be added between each row. The string consists of a space-separated sequence of specifiers, each of which can have the following values: `none`, `solid`, or `dashed`.

columnlines **of type** DOMString A string specifying whether and what kind of lines should be added between each column. The string consists of a space-separated sequence of specifiers, each of which can have the following values: `none`, `solid`, or `dashed`.

frame **of type** DOMString A string specifying a frame around the table. Allowed values are `(none | solid | dashed)`.

framespacing **of type** DOMString A string of the form `number h-unit number v-unit` specifying the spacing between table and its frame.

equalrows **of type** DOMString A string with the values `true` or `false`.

equalcolumns **of type** DOMString A string with the values `true` or `false`.

displaystyle **of type** DOMString A string with the values `true` or `false`.

side **of type** DOMString A string with the values `left`, `right`, `leftoverlap`, or `rightoverlap`.

minlabelspacing **of type** DOMString A string of the form `number h-unit`, specifying the minimum space between a label and the adjacent entry in the labeled row.

rows **of type** MathMLDocumentFragment**, readonly** A `MathMLDocumentFragment` consisting of `MathMLTableRowElements` and `MathMLLabeledRowElements` representing the rows of the table. This is a *live* object.

Methods

insertEmptyRow

A convenience method to insert a new (empty) row (`mtr`) in the table before the current `index`th row. If `index` is less than 0, the new row is inserted before the `-index`th row counting up from the current last row; if `index` is equal to the current number of rows, the new row is appended as the last row.

Parameters

`long` `index` Position before which to insert the new row, where 0 represents the first row. Negative numbers are used to count backwards from the last row

Return value

`MathMLTableRowElement` Returns the `MathMLTableRowElement` child of this `MathMLTableElement` that represents the new `mtr` element being inserted

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the current number of rows of this `mtable` element or less than minus this number.

insertEmptyLabeledRow

A convenience method to insert a new (empty) labeled row (`mlabeledtr`) in the table before the current `index`th row. If `index` is less than 0, the new row is inserted before the `-index`th row counting up from the current last row; if `index` is equal to the current number of rows, the new row is appended as the last row.

Parameters

`long` `index` Position before which to insert the new row, where 0 represents the first row. Negative numbers are used to count backwards from the last row

Return value

`MathMLLabeledRowElement` Returns the `MathMLLabeledRowElement` child of this `MathMLTableElement` representing the `mtr` element being inserted.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the current number of rows of this `mtable` element or less than minus this number.

getRow

A convenience method to retrieve the `index`th row from the table. If `index` is less than 0, the `-index`th row from the bottom of the table is retrieved. (So, for instance, if `index` is -2, the next-to-last row is retrieved.) If `index` is not a valid value (i.e. is greater than or equal to the number of rows, or is less than minus the number of rows), a `null MathMLTableRowElement` is returned.

Parameters

`unsigned long` `index` Index of the row to be returned, where 0 represents the first row. Negative numbers are used to count backwards from the last row

Return value

`MathMLTableRowElement` Returns the `MathMLTableRowElement` representing the `index`th row of the table.

This method raises no exceptions.

insertRow

A convenience method to insert the new row or labeled row (`mtr` or `mlabeledtr`) represented by `newRow` in the table before the current `index`th row. If `index` is equal to the current number of rows, `newRow` is appended as the last row in the table. If `index` is less than 0, the new row is inserted before the `-index`th row from the bottom of the table. (So, for instance, if `index` is -2, the new row is inserted before the next-to-last current row.)

Parameters

`long`	`index`	Index before which to insert `newRow`, where 0 represents the first row. Negative numbers are used to count backwards
`MathMLTableRowElement`	`newRow`	A `MathMLTableRowElement` or `MathMLLabeledRowElement` representing the row to be inserted.

Return value

`MathMLTableRowElement` The `MathMLTableRowElement` or `MathMLLabeledRowElement` child of this `MathMLTableElement` representing the `mtr` elem

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newRow` is not a `MathMLTableRowElement` or `MathMLLabeledRowElement`.

INDEX_SIZE_ERR Raised if `index` is greater than the current number of rows or less than minus the current number of rows of this `mtable` element.

setRow

A method to set the value of the row in the table at the specified index to the `mtr` or `mlabeledtr` represented by `newRow`. If `index` is less than 0, the `-index`th row counting up from the last is replaced by `newRow`; if `index` is one more than the current number of rows, the new row is appended as the last row in the table.

Parameters

`long`	`index`	Index of the row to be set to `newRow`, where 0 represents the first row. Negative numbers are used to count backwards
`MathMLTableRowElement`	`newRow`	A `MathMLTableRowElement` representing the row that is to be the new `index`th row.

Return value

`MathMLTableRowElement` Returns the `MathMLTableRowElement` or `MathMLLabeledRowElement` child of this element that represents the new row in the

Exceptions

HIERARCHY_REQUEST_ERR Raised if `newRow` is not a `MathMLTableRowElement` or `MathMLLabeledRowElement`.

INDEX_SIZE_ERR Raised if `index` is greater than the current number of rows of this `mtable` element or less than minus this number.

deleteRow

A convenience method to delete the row of the table at the specified index. If `index` is less than 0, the `-index`th row from the bottom of the table is deleted. (So, for instance, if `index` is -2, the next-to-last row is deleted.)

Parameters

`unsigned long` `index` Index of row to be deleted, where 0 represents the first row.

Return value

`void` None.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than or equal to the current number of rows of this `mtable` element or less than minus this number.

removeRow

A convenience method to delete the row of the table at the specified index and return it to the caller. If `index` is less than 0, the `-index`th row from the bottom of the table is deleted. (So, for instance, if `index` is -2, the next-to-last row is deleted.)

Parameters

`long` `index` Index of row to be removed, where 0 represents the first row.

Return value

`MathMLTableRowElement` A `MathMLTableRowElement` representing the row being deleted.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than or equal to the number of rows of this `mtable` element or less than minus this number.

Interface MathMLTableRowElement

Extends: MathMLPresentationElement

This interface extends the `MathMLPresentationElement` interface for the MathML table or matrix row element `mtr`.

IDL Definition

```
interface MathMLTableRowElement: MathMLPresentationElement {
  attribute DOMString rowalign;
  attribute DOMString columnalign;
  attribute DOMString groupalign;
  readonly attribute MathMLDocumentFragment cells;
  MathMLTableCellElement insertEmptyCell(in unsigned long index);
  MathMLTableCellElement insertCell(in MathMLTableCellElement newCell, in unsigned long index);
  MathMLTableCellElement setCell(in MathMLTableCellElement newCell, in unsigned long index);
  void deleteCell(in unsigned long index);
};
```

Attributes

rowalign **of type** `DOMString` A string representing an override of the row alignment specified in the containing `mtable`. Allowed values are `top`, `bottom`, `center`, `baseline`, and `axis`.

columnalign **of type** `DOMString` A string representing an override of the column alignment specified in the containing `mtable`. Allowed values are `left`, `center`, and `right`.

groupalign **of type** `DOMString` [To be changed?]

cells **of type** `MathMLDocumentFragment`**, readonly** A `MathMLDocumentFragment` consisting of the cells of the row. *Note that this does not include the label if this is a `MathMLLabeledRowElement`!*

Methods

insertEmptyCell

A convenience method to insert a new (empty) cell in the row.

Parameters

`unsigned long`	`index`	Index of the cell before which the new cell is to be inserted, where the first cell is numbered 0. If `index` is equal to the curre new cell is appended as the last cell of the row. *Note that the index will differ from the index of the corresponding `Node` in the `Node::childNodes` if this is a `MathMLLabeledRowElement`!*

Return value

`MathMLTableCellElement`	Returns the `MathMLTableCellElement` representing the `mtd` element being inserted.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the current number of cells of this `mtr` element.

insertCell

A convenience method to insert a new cell in the row.

Parameters

`MathMLTableCellElement`	`newCell`	A `MathMLTableCellElement` representing the new cell (`mtd` element) to be inserted.
`unsigned long`	`index`	Index of the cell before which the new cell is to be inserted, where the first cell is numbered 0. If `index` equals cell is appended as the last cell of the row. *Note that the index will differ from the index of the corresponding No `MathMLLabeledRowElement`!*

Return value

MathMLTableCellElement The `MathMLTableCellElement` representing the `mtd` element being inserted.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the current number of cells of this `mtr` element.

setCell

A convenience method to set the value of a cell in the row to `newCell`. If index is equal to the current number of cells, `newCell` is appended as the last cell in the row.

Parameters

MathMLTableCellElement	newCell	A `MathMLTableCellElement` representing the cell (`mtd` element) that is to be inserted.
unsigned long	index	Index of the cell that is to be replaced by the new cell, where the first cell is numbered 0. *Note that the index will diff* *`Node` in the collection returned by `Node::childNodes` if this is a `MathMLLabeledRowElement`!*

Return value

MathMLTableCellElement The `MathMLTableCellElement` child of this `MathMLTableRowElement` representing the new `mtd` element.

This method raises no exceptions.

deleteCell

A convenience method to delete a cell in the row.

Parameters

unsigned long index Index of cell to be deleted. *Note that the count will differ from the `index`th child node if this is a `MathMLLabeledRowElement`!*

Return value

void None.

This method raises no exceptions.

Interface MathMLLabeledRowElement

Extends: MathMLTableRowElement

This interface extends the `MathMLTableRowElement` interface to represent the `mlabeledtr` element Section 3.5.3. *Note that the presence of a label causes the `index`th child node to differ from the `index`th cell!*

IDL Definition

```
interface MathMLLabeledRowElement: MathMLTableRowElement {
  attribute MathMLElement label;
};
```

Attributes

label of type MathMLElement A `MathMLElement` representing the label of this row. Note that retrieving this should have the same effect as a call to `Node::getfirstChil` while setting it should have the same effect as `Node::replaceChild(Node::getfirstChild())`.

NO_MODIFICATION_ALLOWED_ERR Raised if this `MathMLElement` or the new `MathMLElement` is read-only.

Interface MathMLTableCellElement

Extends: MathMLPresentationContainer

This interface extends the `MathMLPresentationContainer` interface for the MathML table or matrix cell element `mtd`.

IDL Definition

```
interface MathMLTableCellElement: MathMLPresentationContainer {
  attribute DOMString rowspan;
  attribute DOMString columnspan;
  attribute DOMString rowalign;
  attribute DOMString columnalign;
  attribute DOMString groupalign;
  readonly attribute boolean hasaligngroups;
  readonly attribute DOMString cellindex;
};
```

Attributes

rowspan **of type** `DOMString` A string representing a positive integer that specifies the number of rows spanned by this cell. The default is 1.

columnspan **of type** `DOMString` A string representing a positive integer that specifies the number of columns spanned by this cell. The default is 1.

rowalign **of type** `DOMString` A string specifying an override of the inherited vertical alignment of this cell within the table row. Allowed values are `top`, `bottom`, `center`, `baseline`, and `axis`.

columnalign **of type** `DOMString` A string specifying an override of the inherited horizontal alignment of this cell within the table column. Allowed values are `left`, `center`, and `right`.

groupalign **of type** `DOMString` A string specifying how the alignment groups within the cell are to be aligned with those in cells above or below this cell. The string consists of a space-separated sequence of specifiers, each of which can have the following values: `left`, `right`, `center`, or `decimalpoint`.

hasaligngroups **of type** `boolean`**, readonly** A string with the values `true` or `false` indicating whether the cell contains align groups.

cellindex **of type** `DOMString`**, readonly** A string representing the integer index (1-based?) of the cell in its containing row. [What about spanning cells? How do these affect this value?]

Interface MathMLAlignGroupElement

Extends: `MathMLPresentationElement`

This interface extends the `MathMLPresentationElement` interface for the MathML group -alignment element `<maligngroup/>`.

IDL Definition

```
interface MathMLAlignGroupElement: MathMLPresentationElement {
  attribute DOMString groupalign;
};
```

Attributes

groupalign **of type** `DOMString` A string specifying how the alignment group is to be aligned with other alignment groups above or below it. Allowed values are `left`, `right`, `center`, or `decimalpoint`.

Interface MathMLAlignMarkElement

Extends: `MathMLPresentationElement`

This interface extends the `MathMLPresentationElement` interface for the MathML *alignment mark* element `<malignmark/>`.

IDL Definition

```
interface MathMLAlignMarkElement: MathMLPresentationElement {
  attribute DOMString edge;
};
```

Attributes

edge of type `DOMString` A string specifying alignment on the right edge of the preceding element or the left edge of the following element. Allowed values are `left` and `right`.

E.1.4 Content Elements

Interface MathMLContentElement

Extends: `MathMLElement`

This interface is provided to serve as a base interface for various MathML Content interfaces. It contains no new attributes or methods at this time; however, it is felt that the distinction between Presentation and Content MathML entities should be indicated in the MathMLElement heirarchy. In particular, future versions of the MathML DOM may add functionality on this interface; it may also serve as an aid to implementors.

IDL Definition

```
interface MathMLContentElement: MathMLElement {
};
```

E.1.4.1 Content Token Interfaces

Interface MathMLContentToken

Extends: `MathMLContentElement`

This interface represents the MathML Content token elements: `ci`, `cn` and `csymbol`. These elements may contain MathML Presentation elements, `Text` nodes, or a combination of both. Thus the `getArgument` and `insertArgument` methods have been provided to deal with this distinction between these elements and other MathML Content elements.

IDL Definition

```
interface MathMLContentToken: MathMLContentElement {
  readonly attribute MathMLDocumentFragment arguments;
  Node getArgument(in unsigned long index);
  Node insertArgument(in unsigned long index, in Node newArgument);
  Node setArgument(in unsigned long index, in Node newArgument);
  void deleteArgument(in unsigned long index);
  Node removeArgument(in unsigned long index);
};
```

Attributes

arguments of type `MathMLDocumentFragment`**, readonly** The arguments of this element, returned as a `MathMLDocumentFragment`. Note that this is not necessarily the same as `Node::childNodes`, particularly in the case of the `cn` element. The reason is that the `sep` elements that are used to separate the arguments of a `cn` are not returned.

Methods

getArgument

A convenience method to retrieve the child argument at the position referenced by `index`. Note that this is not necessarily the same as the `index`th child `Node` of this `Element`; in particular, `sep` elements will not be counted.

Parameters

`unsigned long`	`index`	Position of desired argument in the list of arguments. The first argument is numbered 1.

Return value

`Node`	The `Node` retrieved.

This method raises no exceptions.

insertArgument

A convenience method to insert `newArgument` before the current `index`th argument child of this element. If `index` is 0, `newArgument` is appended as the last argument.

Parameters

`unsigned long`	`index`	Position before which `newArgument` is to be inserted. The first argument is numbered 1.*Note that this is not necessarily th child nodes, as nodes representing such elements as `sep` are not counted as arguments.*
`Node`	`newArgument`	`Node` to be inserted as the `index`th argument. This will either be a `MathMLElement` or a `Text` node.

Return value

`Node`	The `Node` inserted. This is the element within the DOM.

This method raises no exceptions.

setArgument

A convenience method to set an argument child at the position referenced by `index`. If there is currently an argument at this position, it is replaced by `newArgument`.

Parameters

`unsigned long`	`index`	Position of the argument that is to be set to `newArgument` in the list of arguments. The first argument is numbered 1. *Not index of the `Node` in the list of child nodes, as nodes representing such elements as `sep` are not counted as arguments.*
`Node`	`newArgument`	`Node` to be inserted as the argument. This will either be a `MathMLElement` or a `Text` node.

Return value

`Node`	The `Node` inserted. This is the element within the DOM.

This method raises no exceptions.

deleteArgument

A convenience method to delete the argument child located at the position referenced by `index`.

Parameters

`unsigned long`	`index`	Position of the argument to be deleted from the list of arguments. The first argument is numbered 1.

Return value

`void`	None.

This method raises no exceptions.

removeArgument

A convenience method to delete the argument child located at the position referenced by `index`, and to return it to the caller.

Parameters

`unsigned long`	`index`	Position of the argument to be deleted from the list of arguments. The first argument is numbered 1.

Return value

`Node`	A `Node` representing the deleted argument.

This method raises no exceptions.

Interface MathMLcnElement

Extends: `MathMLContentToken`

The `cn` element is used to specify actual numeric constants.

IDL Definition

```
interface MathMLcnElement: MathMLContentToken {
  attribute DOMString type;
  attribute DOMString base;
  readonly attribute unsigned long nargs;
  attribute DOMString definitionURL;
  attribute DOMString encoding;
};
```

Attributes

type **of type** `DOMString` Values include, but are not restricted to, `center`, `real`, `integer`, `rational`, `complex-cartesian`, `complex-polar`, and `constant`.

base **of type** `DOMString` A string representing an integer between 2 and 36; the base of the numerical representation.

nargs **of type** `unsigned long`**, readonly** The number of `sep`-separated arguments.

definitionURL **of type** `DOMString` A URL pointing to an alternative definition

encoding **of type** `DOMString` A description of the syntax used in `definitionURL`.

Interface MathMLciElement

Extends: `MathMLContentToken`

The `ci` element is used to specify a symbolic name.

IDL Definition

```
interface MathMLciElement: MathMLContentToken {
  attribute DOMString type;
};
```

Attributes

type **of type** `DOMString` Values include `integer`, `rational`, `real`, `float`, `complex`, `complex-polar`, `complex-cartesian`, `constant`, any of the MathML content container types (`vector`, `matrix`, `set`, `list` etc.) or their types.

Interface MathMLcsymbolElement

Extends: `MathMLContentToken`

This interface represents the `csymbol` element.

IDL Definition

```
interface MathMLcsymbolElement: MathMLContentToken {
  attribute DOMString definitionURL;
  attribute DOMString encoding;
};
```

Attributes

definitionURL **of type** `DOMString` A URI pointing to a definition for this symbol element.

encoding **of type** `DOMString` A string describing the syntax in which the definition located at `definitionURL` is given.

E.1.4.2 Content Container Interfaces

We have added interfaces for content elements that are containers, i.e. elements that may contain child elements corresponding to arguments, bound variables, conditions, or lower or upper limits.

Interface MathMLContentContainer

Extends: `MathMLContentElement`

This interface supports the MathML Content elements that may contain child Content elements. They include: `apply`, `reln` (deprecated), `fn`, `lambda`, `condition`, `declare`, `semantics`, `annotation`, `annotation-xml`, `interval`, `set`, `list`, `vector`, `matrix`, `matrixrow`, `lowlimit`, `uplimit`, `bvar`, `degree`, `set`, `list`, and `logbase`.

IDL Definition

```
interface MathMLContentContainer: MathMLContentElement {
  readonly attribute unsigned long nBoundVariables;
  readonly attribute MathMLDocumentFragment nArguments;
  attribute MathMLconditionElement condition;
  attribute MathMLElement lowLimit;
  attribute MathMLElement upLimit;
  attribute MathMLElement opDegree;
  MathMLElement getArgument(in unsigned long index);
  MathMLElement insertArgument(in MathMLElement newArgument, in unsigned long index);
  MathMLElement setArgument(in MathMLElement newArgument, in unsigned long index);
  void deleteArgument(in unsigned long index);
  MathMLElement removeArgument(in unsigned long index);
  MathMLbvarElement getBoundVariable(in unsigned long index);
  MathMLbvarElement insertBoundVariable(in MathMLbvarElement newBVar, in unsigned long index);
  MathMLbvarElement setBoundVariable(in MathMLbvarElement newBVar, in unsigned long index);
  void deleteBoundVariable(in unsigned long index);
  MathMLbvarElement removeBoundVariable(in unsigned long index);
};
```

Attributes

nBoundVariables **of type** `unsigned long`**, readonly** The number of `bvar` child elements of this element.

nArguments **of type** `MathMLDocumentFragment`**, readonly** The number of child elements of this element which represent arguments of the element, as opposed to qualifiers. Thus it does not contain elements representing bound variables, conditions, separators, degrees, or upper or lower limits (`bvar`, `condition`, `sep`, `degree`, `lowlimit`, or `uplimit`).

condition **of type** `MathMLconditionElement` This attribute represents the `condition` child element of this node. See Section 4.2.3.2.

HIERARCHY_REQUEST_ERR Raised if this element does not permit a child `condition` element. In particular, raised if this element is not a `apply`, `set`, or `list`.

lowLimit **of type** MathMLElement This attribute represents the `lowlimit` child element of this node (if any). This expresses, for instance, the lower limit of integration if this is an `apply` element whose first child is a `int`. See Section 4.2.3.2.

HIERARCHY_REQUEST_ERR Raised if this element does not permit a child `lowlimit` element. In particular, raised if this element is not an `apply` element whose first child is an `int`, `sum`, `product`, or `limit` element.

upLimit **of type** MathMLElement This attribute represents the `uplimit` child element of this node (if any). This expresses, for instance, the upper limit of integration if this is an `apply` element whose first child is a `int`. See Section 4.2.3.2.

HIERARCHY_REQUEST_ERR Raised if this element does not permit a child `uplimit` element. In particular, raised if this element is not an `apply` element whose first child is an `int`, `sum`, or `product` element.

opDegree **of type** MathMLElement This attribute represents the `degree` child element of this node. This expresses, for instance, the degree of differentiation if this element is a `bvar` child of an `apply` element whose first child is a `diff` or `partialdiff`. See Section 4.2.3.2.

HIERARCHY_REQUEST_ERR Raised if this element does not permit a child `degree` element. In particular, raised if this element is not a `bvar` or `apply`.

Methods

getArgument

This method returns only the child elements that are arguments of this element. *This frequently differs from the value of* `Node::childNodes().item(index)`, as elements representing bound variables, conditions, separators, degrees, and upper or lower limits are not returned (`bvar`, `condition`, `sep`, `degree`, `lowlimit`, `uplimit`).

Parameters

unsigned long	index	The one-based index of the argument to be retrieved.

Return value

MathMLElement	A `MathMLElement` representing the `index`-thargument of this element.

This method raises no exceptions.

insertArgument

This method inserts `newArgument` before the current `index`th argument of this element. If `index` is 0, `newArgument` is appended as the last argument. *This frequently differs from setting the node at* `Node::childNodes().item(index)`, as elements representing bound variables, conditions, separators, degrees, and upper or lower limits are not counted (`bvar`, `condition`, `sep`, `degree`, `lowlimit`, `uplimit`).

Parameters

MathMLElement	newArgument	The `MathMLElement` to be inserted as the `index`th argument of this element.
unsigned long	index	The one-based index of the position before which `newArgument` is to be inserted.

Return value

MathMLElement	The `MathMLElement` being inserted as the `index`th argument of this element.

Exceptions

HIERARCHY_REQUEST_ERR Raised if this element does not permit a child `degree` element. In particular, raised if this element is not a `bvar` or `apply`.

setArgument

This method sets `newArgument` as the `index`th argument of this element. If there is currently an `index`th argument, it is replaced by `newArgument`. *This frequently differs from setting the node at* `Node::childNodes().item(index)`, as elements representing bound variables, conditions, separators, degrees, and upper or lower limits are not counted (`bvar`, `condition`, `sep`, `degree`, `lowlimit`, `uplimit`).

Parameters

MathMLElement	newArgument	The `MathMLElement` that is to be the `index`th argument of this element.
unsigned long	index	The one-based index of the position in the argument list into which `newArgument` is to be inserted.

Return value

MathMLElement	The new `MathMLElement` being inserted as the `index`th argument of this element.

Exceptions

HIERARCHY_REQUEST_ERR Raised if this element does not permit a child `degree` element. In particular, raised if this element is not a `bvar` or `apply`.

deleteArgument

This method deletes the `index`th child element that is an argument of this element. Note that child elements which are qualifier elements are not counted in determining the `index`th argument.

Parameters

`unsigned long` `index` The one-based index of the argument to be deleted.

Return value

`void` None.

This method raises no exceptions.

removeArgument

This method deletes the `index`th child element that is an argument of this element, and returns it to the caller. Note that child elements that are qualifier elements are not counted in determining the `index`th argument.

Parameters

`unsigned long` `index` The one-based index of the argument to be removed.

Return value

`MathMLElement` A `MathMLElement` representing the argument being removed.

This method raises no exceptions.

getBoundVariable

This method retrieves the `index`th `MathMLbvarElement` child of the `MathMLElement`. Note that only `bvar` child elements are counted in determining the `index`th bound variable.

Parameters

`unsigned long` `index` The one-based index into the bound variable children of this element of the `MathMLbvarElement` to be retrieved.

Return value

`MathMLbvarElement` The `MathMLbvarElement` representing the `index`th `bvar` child of this element.

This method raises no exceptions.

insertBoundVariable

This method inserts a `MathMLbvarElement` as a child node before the current `index`th bound variable child of this `MathMLElement`. If `index` is 0, `newBVar` is appended as the last bound variable child. This has the effect of adding a bound variable to the expression this element represents. Note that the new bound variable is inserted as the `index`-th `bvar` child node, not necessarily as the `index`th child node. The point of the method is to allow insertion of bound variables without requiring the caller to calculate the exact order of child qualifier elements.

Parameters

`MathMLbvarElement` `newBVar` A `MathMLbvarElement` representing the `bvar` element being added.

`unsigned long` `index` The one-based index into the bound variable children of this element before which `newBVar` is to be inserted.

Return value

`MathMLbvarElement` The `MathMLbvarElement` being added.

Exceptions

HIERARCHY_REQUEST_ERR Raised if this element does not permit child `bvar` elements.

setBoundVariable

This method sets the `index`th bound variable child of this `MathMLElement` to `newBVar`. This has the effect of setting a bound variable in the expression this element represents. Note that the new bound variable is inserted as the `index`-th `bvar` child node, not necessarily as the `index`th child node. The point of the method is to allow insertion of bound variables without requiring the caller to calculate the exact order of child qualifier elements. If there is already a `bvar` at the `index`-th position, it is replaced by `newBVar`.

Parameters

`MathMLbvarElement` `newBVar` The new `MathMLbvarElement` child of this element being set.

`unsigned long` `index` The one-based index into the bound variable children of this element at which `newBVar` is to be inserted.

Return value

`MathMLbvarElement` The `MathMLbvarElement` being added.

Exceptions

HIERARCHY_REQUEST_ERR Raised if this element does not permit child `bvar` elements.

deleteBoundVariable

This method deletes the `index`-th `MathMLbvarElement` child of the `MathMLElement`. This has the effect of removing this bound variable from the list of qualifiers affecting the element this represents.

Parameters

`unsigned long` `index` The one-based index into the bound variable children of this element of the `MathMLbvarElement` to be removed.

Return value

`void` None.

This method raises no exceptions.

removeBoundVariable

This method removes the `index`-th `MathMLbvarElement` child of the `MathMLElement` and returns it to the caller. This has the effect of removing this bound variable from the list of qualifiers affecting the element this represents.

Parameters

`unsigned long` `index` The one-based index into the bound variable children of this element of the `MathMLbvarElement` to be removed.

Return value

`MathMLbvarElement` The `MathMLbvarElement` being removed.

This method raises no exceptions.

Interface MathMLapplyElement

Extends: `MathMLContentContainer`

The `apply` element allows a function or operator to be applied to its arguments.

IDL Definition

```
interface MathMLapplyElement: MathMLContentContainer {
  attribute MathMLElement operator;
};
```

Attributes

operator **of type** `MathMLElement` The MathML element representing the function or operator that is applied to the list of arguments.

Interface MathMLfnElement

Extends: `MathMLContentContainer`

The `fn` element makes explicit the fact that a more general MathML object is intended to be used in the same manner as if it were a pre-defined function such as `sin` or `plus`.

IDL Definition

```
interface MathMLfnElement: MathMLContentContainer {
  attribute DOMString definitionURL;
  attribute DOMString encoding;
};
```

Attributes

definitionURL **of type** `DOMString` A URL pointing to a definition for this function-type element. Note that there is no stipulation about the form this definition may take!

encoding **of type** `DOMString` A string describing the syntax in which the definition located at `definitionURL` is given.

Interface MathMLlambdaElement

Extends: `MathMLContentContainer`

The `lambda` element is used to construct a user-defined function from an expression and one or more free variables.

IDL Definition

```
interface MathMLlambdaElement: MathMLContentContainer {
  attribute MathMLElement expression;
};
```

Attributes

expression **of type** `MathMLElement` The `MathMLElement` representing the expression. This is included only as a convenience; getting it should give the same result as `MathMLContentContainer::getArgument(1)`.

Interface MathMLsetElement

Extends: `MathMLContentContainer`

The `set` element is the container element that represents a set of elements. The elements of a set can be defined either by explicitly listing the elements, or by using the `bvar` and `condition` elements.

IDL Definition

```
interface MathMLsetElement: MathMLContentContainer {
  readonly attribute boolean isExplicit;
  attribute DOMString type;
};
```

Attributes

isExplicit **of type** `boolean`**, readonly** This is true if the set is specified by giving the list of its elements explicitly.

type **of type** `DOMString` The `type` attribute of the represented element. Predefined values are `norml` and `multiset`. See Section 4.4.6 and Section 4.3.

Interface MathMLlistElement

Extends: `MathMLContentContainer`

The `list` element is the container element which represents a list of elements. Elements can be defined either by explicitly listing the elements, or by using the `bvar` and `condition` elements.

IDL Definition

```
interface MathMLlistElement: MathMLContentContainer {
  readonly attribute boolean isExplicit;
  attribute DOMString ordering;
};
```

Attributes

isExplicit of type `boolean`, readonly This is true if the list is specified by giving its elements explicitly.

ordering of type `DOMString` The `order` attribute of the represented element. Predefined values are `numeric` and `lexicographic`. See Section 4.4.6 and Section 4.3.

Interface MathMLbvarElement

Extends: `MathMLContentContainer`

This interface represents the MathML bound variable element `bvar`. The interface currently provides no functionality beyond that of `MathMLContentContainer`, but is useful for defining the type of bound variable access functions.

IDL Definition

```
interface MathMLbvarElement: MathMLContentContainer {
};
```

E.1.4.3 Content Leaf Element Interfaces

Interface MathMLpredefinedSymbol

Extends: `MathMLContentElement`

This interface supports all of the empty built-in operator, relation, function, and constant and symbol elements that have the `definitionURL` and `encoding` attributes in addition to the standard set of attributes. The elements supported in order of their appearance in Section 4.4 are: `inverse`, `compose`, `ident`, `quotient`, `exp`, `factorial`, `divide`, `max`, `min`, `minus`, `plus`, `power`, `rem`, `times`, `root`, `gcd`, `and`, `or`, `xor`, `not`, `implies`, `forall`, `exists`, `abs`, `conjugate`, `eq`, `neq`, `gt`, `lt`, `geq`, `leq`, `ln`, `log`, `int`, `diff`, `partialdiff`, `union`, `intersect`, `in`, `notin`, `subset`, `prsubset`, `notsubset`, `notprsubset`, `setdiff`, `sum`, `product`, `limit`, `tendsto`, `sin`, `cos`, `tan`, `sec`, `csc`, `cot`, `sinh`, `cosh`, `tanh`, `sech`, `csch`, `coth`, `arcsin`, `arccos`, `arctan`, `mean`, `sdev`, `variance`, `median`, `mode`, `moment`, `determinant`, `transpose`, `integers`, `reals`, `rationals`, `naturalnumbers`, `complexes`, `primes`, `exponentiale`, `imaginaryi`, `notanumber`, `true`, `false`, `emptyset`, `pi`, `eulergamma`, and `infinity`.

IDL Definition

```
interface MathMLpredefinedSymbol: MathMLContentElement {
  attribute DOMString definitionURL;
  attribute DOMString encoding;
  attribute DOMString arity;
  readonly attribute DOMString symbolName;
};
```

Attributes

definitionURL **of type** DOMString A string that provides an override to the default semantics, or provides a more specific definition

encoding **of type** DOMString A string describing the syntax in which the definition located at `definitionURL` is given.

arity **of type** DOMString A string representing the number of arguments. Values include 0, 1, ... and `variable`.

symbolName **of type** DOMString**, readonly** A string giving the name of the MathML element represented. This is a convenience attribute only; accessing it should be synonymous with accessing the `Element::tagName` attribute.

E.1.4.4 Other Content Element Interfaces

Interface MathMLintervalElement

Extends: MathMLContentElement

The `interval` element is used to represent simple mathematical intervals on the real number line. It contains either two child elements that evaluate to real numbers or one child element that is a condition for defining membership in the interval.

IDL Definition

```
interface MathMLintervalElement: MathMLContentElement {
  attribute DOMString closure;
  attribute MathMLcnElement start;
  attribute MathMLcnElement end;
};
```

Attributes

closure **of type** DOMString A string with value `open`, `closed`, `open-closed` or `closed-open`. The default value is `closed`.

start **of type** MathMLcnElement A `MathMLcnElement` representing the real number defining the start of the interval. If `end` has not already been set, it becomes the same as `start` until set otherwise.

end **of type** MathMLcnElement A `MathMLcnElement` representing the real number defining the end of the interval. If `start` has not already been set, it becomes the same as `end` until set otherwise.

Interface MathMLconditionElement

Extends: MathMLContentElement

The `condition` element is used to place a condition on one or more free variables or identifiers.

IDL Definition

```
interface MathMLconditionElement: MathMLContentElement {
  attribute MathMLapplyElement condition;
};
```

Attributes

condition **of type** MathMLapplyElement A `MathMLapplyElement` that represents the condition.

Interface MathMLdeclareElement

Extends: MathMLContentElement

The declare construct has two primary roles. The first is to change or set the default attribute values for a specific mathematical object. The second is to establish an association between a 'name' and an object.

IDL Definition

```
interface MathMLdeclareElement: MathMLContentElement {
  attribute DOMString type;
  attribute unsigned long nargs;
  attribute DOMString occurrence;
  attribute DOMString definitionURL;
  attribute DOMString encoding;
  attribute MathMLciElement identifier;
  attribute MathMLElement constructor;
};
```

Attributes

type **of type** DOMString A string indicating the type of the identifier. It must be compatible with the type of the constructor, if a constructor is present. The type is inferred from the constructor if present, otherwise it must be specified.

nargs **of type** unsigned long If the identifier is a function, this attribute specifies the number of arguments the function takes. This represents the declare element's nargs attribute; see Section 4.4.2.8.

occurrence **of type** DOMString A string with the values prefix, infix, postfix, or function-model.

definitionURL **of type** DOMString A URL specifying the detailed semantics of the element.

encoding **of type** DOMString A description of the syntax used in definitionURL.

identifier **of type** MathMLciElement A MathMLciElement representing the name being declared.

constructor **of type** MathMLElement An optional MathMLElement providing an initial value for the object being declared.

Interface MathMLvectorElement

Extends: MathMLContentElement

vector is the container element for a vector.

IDL Definition

```
interface MathMLvectorElement: MathMLContentElement {
  readonly attribute unsigned long ncomponents;
  MathMLContentElement getComponent(in unsigned long index);
  MathMLContentElement insertComponent(in MathMLContentElement newComponent, in unsigned long index);
  MathMLContentElement setComponent(in MathMLContentElement newComponent, in unsigned long index);
   deleteComponent(in unsigned long index);
  MathMLContentElement removeComponent(in unsigned long index);
};
```

Attributes

ncomponents of type `unsigned long`, readonly The number of components in the vector.

Methods

getComponent

A convenience method to retrieve a component.

Parameters

`unsigned long`	`index`	Position of the component in the list of components. The first element is numbered 1.

Return value

`MathMLContentElement`	The `MathMLContentElement` component at the position specified by `index`. If `index` is not a valid index (i.e. is greater than the of the vector or less than 1), a `null MathMLContentElement` is returned.

This method raises no exceptions.

insertComponent

A convenience method to insert a new component in the vector before the current `index`th component. If `index` is 0 or is one more than the number of components currently in the vector, `newComponent` is appended as the last component of the vector.

Parameters

`MathMLContentElement`	`newComponent`	A `MathMLContentElement` representing the component that is to be added.
`unsigned long`	`index`	Position of the component in the list of components. The first component is numbered 1.

Return value

`MathMLContentElement`	The `MathMLContentElement` child of this `MathMLvectorElement` representing the new component in the DOM.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than one more than the current number of components of this `vector` element.

setComponent

A convenience method to set the `index`th component of the vector to `newComponent`. If `index` is one more than the current number of components, `newComponent` is appended as the last component.

Parameters

`MathMLContentElement`	`newComponent`	A `MathMLContentElement` representing the element that is to be the `index`-th component of the vector.
`unsigned long`	`index`	Position of the component in the list of components. The first element is numbered 1.

Return value

`MathMLContentElement`	The `MathMLContentElement` child of this `MathMLvectorElement` that represents the new component in the DOM.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than one more than the current number of components of this `vector` element.

deleteComponent

A convenience method to delete an element. The deletion changes the indices of the following components.

Parameters

`unsigned long`	`index`	Position of the component in the vector. The position of the first component is 1

Return value

None

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the current number of components of this `vector` element.

removeComponent

A convenience method to remove a component from a vector and return it to the caller. If index is greater than the number of components or is 0, a `null MathMLContentElement` is returned.

Parameters

`unsigned long`	`index`	Position of the component in the list of components. The first element is numbered 1.

Return value

MathMLContentElement The MathMLContentElement component being removed.
This method raises no exceptions.

Interface MathMLmatrixElement

Extends: MathMLContentElement

The matrix element is the container element for matrixrow elements.

IDL Definition

```
interface MathMLmatrixElement: MathMLContentElement {
  readonly attribute unsigned long nrows;
  readonly attribute unsigned long ncols;
  readonly attribute MathMLDocumentFragment rows;
  MathMLmatrixrowElement getRow(in unsigned long index);
  MathMLmatrixrowElement insertRow(in MathMLmatrixrowElement newRow, in unsigned long index);
  MathMLmatrixrowElement setRow(in MathMLmatrixrowElement newRow, in unsigned long index);
   deleteRow(in unsigned long index);
   removeRow(in unsigned long index);
};
```

Attributes

nrows of type unsigned long**, readonly** The number of rows in the represented matrix.
ncols of type unsigned long**, readonly** The number of columns in the represented matrix.
rows of type MathMLDocumentFragment**, readonly** The rows of the matrix, returned as a MathMLDocumentFragment consisting of MathMLmatrixrowElements.

Methods

getRow

A convenience method to retrieve a specified row.
Parameters
unsigned long index Position of the row in the list of rows. The first row is numbered 1.
Return value
MathMLmatrixrowElement The MathMLmatrixrowElement representing the index-th row.
Exceptions
INDEX_SIZE_ERR Raised if index is greater than the number of rows in the matrix.

insertRow

A convenience method to insert a row before the row that is currently the indexth row of this matrix. If index is 0, newRow is appended as the last row of the matrix.
Parameters
MathMLmatrixrowElement newRow MathMLmatrixrowElement to be inserted into the matrix.
unsigned long index Unsigned integer giving the row position before which newRow is to be inserted. The first row is numbered 1.
Return value
MathMLmatrixrowElement The MathMLmatrixrowElement added. This is the new element within the DOM.
Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than one more than the number of rows in the matrix.

HIERARCHY_REQUEST_ERR Raised if the number of cells in `newRow` doesn't match the number of columns in the matrix.

setRow

A convenience method to set the value of the `index`th child `matrixrow` element of this element. If there is already a row at the specified index, it is replaced by `newRow`.

Parameters

`MathMLmatrixrowElement`	`newRow`	`MathMLmatrixrowElement` representing the `matrixrow` which is to become the `index`th row of the matrix.
`unsigned long`	`index`	Unsigned integer giving the row which is to be set to `newRow`. The first row is numbered 1.

Return value

`MathMLmatrixrowElement` The `MathMLmatrixrowElement` child of this `MathMLmatrixrowElement` representing `newRow` within the DOM.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of rows in the matrix.

HIERARCHY_REQUEST_ERR Raised if the number of cells in `newRow` doesn't match the number of columns in the matrix.

deleteRow

A convenience method to delete a row. The deletion changes the indices of the following rows.

Parameters

`unsigned long` `index` Position of the row to be deleted in the list of rows

Return value

None

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of rows in the matrix.

removeRow

A convenience method to remove a row and return it to the caller. The deletion changes the indices of the following rows.

Parameters

`unsigned long` `index` Position of the row to be removed in the list of rows. The first row is numbered 1.

Return value

The `MathMLmatrixrowElement` being removed.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of rows in the matrix.

Interface MathMLmatrixrowElement

Extends: `MathMLContentElement`

The `matrixrow` element is the container element for the elements of a `matrix`.

IDL Definition

```
interface MathMLmatrixrowElement: MathMLContentElement {
  readonly attribute unsigned long nEntries;
  MathMLContentElement getEntry(in unsigned long index);
  MathMLContentElement insertEntry(in MathMLContentElement newEntry, in unsigned long index);
  MathMLContentElement setEntry(in MathMLContentElement newEntry, in unsigned long index);
   deleteEntry(in unsigned long index);
  MathMLContentElement removeEntry(in unsigned long index);
};
```

Attributes

nEntries of type `unsigned long`**, readonly** The number of entries in the row.

Methods

getEntry

A convenience method to retrieve the contents of an entry by index.

Parameters

`unsigned long` `index` Position of the entry in the row. The first entry is numbered 1.

Return value

`MathMLContentElement` The `MathMLContentElement` element representing the `index`th entry in the row.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of entries in the row.

insertEntry

A convenience method to insert an entry before the current `index`th entry of the row. If `index` is 0, `newEntry` is appended as the last entry. Note that this method increases the size of the `matrixrow`.

Parameters

`MathMLContentElement` `newEntry` The `MathMLContentElement` to be representing the new entry to be inserted into the row.

`unsigned long` `index` The index before which `newEntry` is to be inserted in the row. The first entry is numbered 1.

Return value

`MathMLContentElement` The `MathMLContentElement` child of this `MathMLmatrixrowElement` representing `newEntry` in the DOM.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of entries in the row.

setEntry

A convenience method to set the contents of the entry at position `index` in the row to `newEntry`. If there is already a entry at the specified index, it is replaced by the new entry.

Parameters

`MathMLContentElement` `newEntry` The `MathMLContentElement` representing the element that is to be the `index`th entry.

`unsigned long` `index` The index of the entry that is to be set equal to `newEntry`. The first entry is numbered 1.

Return value

`MathMLContentElement` The `MathMLContentElement` child of this `MathMLmatrixRowElement` representing `newEntry` in the DOM.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than one more than the number of elements in the row.

deleteEntry

A convenience method to delete an entry. The deletion changes the indices of the following entries.

Parameters

`unsigned long` `index` Position of the entry to be deleted in the row. The first entry is numbered 1.

Return value

None

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of entries in the row.

removeEntry

A convenience method to remove an entry from the row and return the removed entry to the caller.

Parameters

`unsigned long` `index` Position of the entry to be removed from the row. The first entry is numbered 1.

Return value

`MathMLContentElement` The `MathMLContentElement` being removed from the row.

Exceptions

INDEX_SIZE_ERR Raised if `index` is greater than the number of entries in the row.

Appendix F

Glossary (Non-Normative)

Several of the following definitions of terms have been borrowed or modified from similar definitions in documents originating from W3C or standards organisations. See the individual definitions for more information.

Argument A child of a presentation layout schema. That is, 'A is an argument of B' means 'A is a child of B and B is a presentation layout schema'. Thus, token elements have no arguments, even if they have children (which can only be `malignmark`).

Attribute A parameter used to specify some property of an SGML or XML element type. It is defined in terms of an attribute name, attribute type, and a default value. A value may be specified for it on a start-tag for that element type.

Axis The axis is an imaginary alignment line upon which a fraction line is centered. Often, operators as well as characters that can stretch, such as parentheses, brackets, braces, summation signs etcetera, are centered on the axis, and are symmetric with respect to it.

Baseline The baseline is an imaginary alignment line upon which a glyph without a descender rests. The baseline is an intrinsic property of the glyph (namely a horizontal line). Often baselines are aligned (joined) during typesetting.

Black box The bounding box of the actual size taken up by the viewable portion (ink) of a glyph or expression.

Bounding box The rectangular box of smallest size, taking into account the constraints on boxes allowed in a particular context, which contains some specific part of a rendered display.

Box A rectangular plane area considered to contain a character or further sub-boxes, used in discussions of rendering for display. It is usually considered to have a baseline, height, depth and width.

Cascading Style Sheets (CSS) A mechanism that allows authors and readers to attach style (e.g. fonts, colors and spacing) to HTML and XML documents.

Character A member of a set of identifiers used for the organization, control or representation of text. ISO/IEC Standard 10646-1:1993 uses the word 'data' here instead of 'text'.

Character data (`CDATA`) A data type in SGML and XML for raw data that does not include markup or entity references. Attributes of type `CDATA` may contain entity references. These are expanded by an XML processor before the attribute value is processed as `CDATA`.

Character or expression depth Distance between the baseline and bottom edge of the character glyph or expression. Also known as the descent.

Character or expression height Distance between the baseline and top edge of the character glyph or expression. Also know as the ascent.

Character or expression width Horizontal distance taken by the character glyph as indicated in the font metrics, or the total width of an expression.

Condition A MathML content element used to place a mathematical condition on one or more variables.

Contained (element A is contained in element B) A is part of B's content.

Container (Constructor) A non-empty MathML Content element that is used to construct a mathematical object such as a number, set, or list.

Content elements MathML elements that explicitly specify the mathematical meaning of a portion of a MathML expression (defined in Chapter 4).

Content token element Content element having only `PCDATA`, `sep` and presentation expressions as content. Represents either an identifier (`ci`) or a number (`cn`).

Context (of a given MathML expression) Information provided during the rendering of some MathML data to the rendering process for the given MathML expression; especially information about the MathML markup surrounding the expression.

Declaration An instance of the declare element.

Depth (of a box) The distance from the baseline of the box to the bottom edge of the box.

Direct sub-expression (of a MathML expression 'E') A sub-expression directly contained in E.

Directly contained (element A in element B) A is a child of B (as defined in XML), in other words A is contained in B, but not in any element that is itself contained in B.

Document Object Model A model in which the document or Web page is treated as an object repository. This model is developed by the DOM Working Group (DOM) of the W3C.

Document Style Semantics and Specification Language (DSSSL) A method of specify the formatting and transformation of SGML documents. ISO International Standard 10179:1996.

Document Type Definition (DTD) In SGML or XML, a DTD is a formal definition of the elements and the relationship among the data elements (the structure) for a particular type of document.

Em A font-relative measure encoded by the font. Before electronic typesetting, an `em` was the width of an 'M' in the font. In modern usage, an `em` is either specified by the designer of the font or is taken to be the height (point size) of the font. Em's are typically used for font-relative horizontal sizes.

Ex A font-relative measure that is the height of an 'x' in the font. `ex`s are typically used for font-relative vertical sizes.

Height (of a box) The distance from the baseline of the box to the top edge of the box.

Inferred `mrow` An `mrow` element that is 'inferred' around the contents of certain layout schemata when they have other than exactly one argument. Defined precisely in Section 3.1.5

Embedded object Embedded objects such as Java applets, Microsoft Component Object Model (COM) objects (e.g. ActiveX Controls and ActiveX Document embeddings), and plug-ins that reside in an HTML document.

Embellished operator An operator, including any 'embellishment' it may have, such as superscripts or style information. The 'embellishment' is represented by a layout schema that contains the operator itself. Defined precisely in Section 3.2.4.

Entity reference A sequence of ASCII characters of the form `&name;` representing some other data, typically a non-ASCII character, a sequence of characters, or an external source of data, e.g. a file containing a set of standard entity definitions such as ISO Latin 1.

Extensible Markup Language (XML) A simple dialect of SGML intended to enable generic SGML to be served, received, and processed on the Web.

Fences In typesetting, bracketing tokens like parentheses, braces, and brackets, which usually appear in matched pairs.

Font A particular collection of glyphs of a typeface of a given size, weight and style, for example 'Times Roman Bold 12 point'.

Glyph The actual shape (bit pattern, outline) of a character. ISO/IEC Standard 9541-1:1991 defines a glyph as a recognizable abstract graphic symbol that is independent of any specific design.

Indirectly contained A is contained in B, but not directly contained in B.

Instance of MathML A single instance of the toplevel element of MathML, and/or a single instance of embedded MathML in some other data format.

Inverse function A mathematical function that, when composed with the original function acts like an identity function.

Lambda expression A mathematical expression used to define a function in terms of variables and an expression in those variables.

Layout schema (plural: schemata) A presentation element defined in chapter 3, other than the token elements and empty elements defined there (i.e. not the elements defined in Section 3.2 and Section 3.5.5, or the empty elements `none` and `mprescripts` defined in Section 3.4.7). The layout schemata are never empty elements (though their content may contain nothing in some cases), are always expressions, and all allow any MathML expressions as arguments (except for requirements on argument count, and the requirement for a certain empty element in `mmultiscripts`).

Mathematical Markup Language (MathML) The markup language specified in this document for describing the structure of mathematical expressions, together with a mathematical context.

MathML element An XML element that forms part of the logical structure of a MathML document.

MathML expression (within some valid MathML data) A single instance of a presentation element, except for the empty elements `none` or `mprescripts`, or an instance of `malignmark` within a token element (defined below); or a single instance of certain of the content elements (see Chapter 4 for a precise definition of which ones).

Multi-purpose Internet Mail Extensions (MIME) A set of specifications that offers a way to interchange text in languages with different character sets, and multi-media content among many different computer systems that use Internet mail standards.

Operator, content element A mathematical object that is applied to arguments using the `apply` element.

Operator, an `mo` element Used to represent ordinary operators, fences, separators in MathML presentation. (The token element`mo` is defined in Section 3.2.4).

OpenMath A general representation language for communicating mathematical objects between application programs.

Parsed character data (`PCDATA`) An SGML/XML data type for raw data occurring in a context where text is parsed and markup (for instance entity references and element start/end tags) is recognised.

Point Point is often abbreviated 'pt'. The value of 1 pt is approximately 1/72 inch. Points are typically used to specify absolute sizes for font-related objects.

Pre-defined function One of the empty elements defined in Section 4.2.3 and used with the `apply` construct to build function applications.

Presentation elements MathML tags and entities intended to express the syntactic structure of mathematical notation (defined in Chapter 3).

Presentation layout schema A presentation element that can have other MathML elements as content.

Presentation token element A presentation element that can contain only parsed character data or the `malignmark` element.

Qualifier A MathML content element that is used to specify the value of a specific named parameter in the application of selected pre-defined functions.

Relation A MathML content element used to construct expressions such as $a < b$.

Render Faithfully translate into application-specific form allowing native application operations to be performed.

Schema Schema (plural: schemata). See 'presentation layout schema'.

Scope of a declaration The portion of a MathML document in which a particular definition is active.

Selected sub-expression (of an `maction` element) The argument of an `maction` element (a layout schema defined in Section 3.6) that is (at any given time) 'selected' within the viewing state of a MathML renderer, or by the `selection` attribute when the element exists only in MathML data. Defined precisely in the abovementioned section.

Space-like (MathML expression) A MathML expression that is ignored by the suggested rendering rules for MathML presentation elements when they determine operator forms and effective operator rendering attributes based on operator positions in `mrow` elements. Defined precisely in Section 3.2.6.

Standard Generalized Markup Language (SGML) An ISO standard (ISO 8879:1986) that provides a formal mechanism for the definition of document structure via DTDs (Document Type Definitions), and a notation for the markup of document instances conforming to a DTD.

Sub-expression (of a MathML expression 'E') A MathML expression contained (directly or indirectly) in the content of E.

Suggested rendering rules for MathML presentation elements Defined throughout Chapter 3; the ones that use other terms defined here occur mainly in Section 3.2.4 and in Section 3.6.

TEX A software system developed by Professor Donald Knuth for typesetting documents.

Token element Presentation token element or a Content token element. (See above.)

Top-level element (of MathML) `math` (defined in Chapter 7).

Typeface A typeface is a specific design of a set of letters, numbers and symbols, such as 'Times Roman' or 'Chicago'.

Valid MathML data MathML data that (1) conforms to the MathML DTD, (2) obeys the additional rules defined in the MathML standard for the legal contents and attribute values of each MathML element, and (3) satisfies the EBNF grammar for content elements.

Width (of a box) The distance from the left edge of the box to the right edge of the box.

Extensible Style Language (XSL) A style language for XML developed by W3C. See XSL FO and XSLT.

XSL Formatting Objects (XSL FO) An XML vocabulary to express formatting, which is a part of XSL.

XSL Transformation (XSLT) A language to express the transformation of XML documents into other XML documents.

Appendix G

Working Group Membership (Non-Normative)

The W3C Math Working Group is presently co-chaired by Patrick Ion of the AMS, and Angel Diaz of IBM. Contact the co-chairs if you are interested in joining the group. For the present membership see its working group home page.

Members of the Working Group responsible for MathML 2.0 are:

- Ron Ausbrooks, Mackichan Software, Las Cruces NM, USA
- Laurent Bernardin, Waterloo Maple, Inc., Waterloo ON, CAN
- Stephen Buswell, Stilo Technologies, Cardiff, UK
- David Carlisle, NAG Ltd., Oxford, UK
- Stéphane Dalmas, INRIA, Sophia Antipolis, FR
- Stan Devitt, Radical Flow Inc., Waterloo ON, CAN
- Angel Diaz, IBM Research Division, Yorktown Heights NY, USA
- Ben Hinkle, Waterloo Maple, Inc., Waterloo ON, CAN
- Stephen Hunt, MATH.EDU Inc., Champaign IL, USA
- Douglas Lovell, IBM Hawthorn Research, Yorktown Heights NY, USA
- Patrick Ion, Mathematical Reviews (American Mathematical Society), Ann Arbor MI, USA
- Robert Miner, Geometry Technologies Inc., Minneapolis MN, USA
- Ivor Philips, Boeing, Seattle WA, USA
- Nico Poppelier, Penta Scope, Amersfoort, NL
- Dave Raggett, W3C (Hewlett Packard), Bristol, UK
- T.V. Raman, IBM Almaden, Palo Alto CA, USA
- Murray Sargent III, Microsoft, Redmond WA, USA
- Neil Soiffer, Wolfram Research Inc., Champaign IL, USA
- Irene Schena, Universitá di Bologna, Bologna, IT
- Paul Topping, Design Science Inc., Long Beach CA, USA
- Stephen Watt, University of Western Ontario, London ON, CAN

Earlier active members of this second W3C Math Working Group have included:

- Sam Dooley, IBM Research, Yortown Heights NY, USA
- Robert Sutor, IBM Research, Yortown Heights NY, USA
- Barry MacKichan, MacKichan Software, Las Cruces NM, USA

At the time of release of MathML 1.0 the Math Working Group was co-chaired by Patrick Ion and Robert Miner, then of the Geometry Center. Since that time several changes in membership have taken place. In the course of the update to MathML 1.01, in addition to people listed in the original membership below, corrections were offered by David Carlisle, Don Gignac, Kostya Serebriany, Ben Hinkle, Sebastian Rahtz, Sam Dooley and others.

Members of the Math Working Group responsible for the finished MathML 1.0 Specification were:

- Stephen Buswell, Stilo Technologies, Cardiff, UK
- Stéphane Dalmas, INRIA, Sophia Antipolis, FR
- Stan Devitt, Maplesoft Inc., Waterloo ON, CAN
- Angel Diaz, IBM Research Division, Yorktown Heights NY, USA
- Brenda Hunt, Wolfram Research Inc., Champaign IL, USA
- Stephen Hunt, Wolfram Research Inc., Champaign IL, USA
- Patrick Ion, Mathematical Reviews (American Mathematical Society), Ann Arbor MI, USA
- Robert Miner, Geometry Center, University of Minnesota, Minneapolis MN, USA
- Nico Poppelier, Elsevier Science, Amsterdam, NL
- Dave Raggett, W3C (Hewlett Packard), Bristol, UK
- T.V. Raman, Adobe Inc., Mountain View CA, USA
- Bruce Smith, Wolfram Research Inc., Champaign IL, USA
- Neil Soiffer, Wolfram Research Inc., Champaign IL, USA
- Robert Sutor, IBM Research, Yorktown Heights NY, USA
- Paul Topping, Design Science Inc., Long Beach CA, USA
- Stephen Watt, University of Western Ontario, London ON, CAN
- Ralph Youngen, American Mathematical Society, Providence RI, USA

Others who had been members of the W3C Math WG for periods at earlier stages were:

- Stephen Glim, Mathsoft Inc., Cambridge MA, USA
- Arnaud Le Hors, W3C, Cambridge MA, USA
- Ron Whitney, Texterity Inc., Boston MA, USA
- Lauren Wood, Softquad, Surrey BC, CAN
- Ka-Ping Yee, University of Waterloo, Waterloo ON, CAN

Appendix H

Changes (Non-Normative)

This appendix summarises the changes with respect to the preceding version (1.01) of the MathML Specification.

- changes to Chapter 1 (upto revision 1.20)
  - rewritten to reflect developments since publication of the MathML 1.0 Recommendation, for example XML, XSL, CSS and schemas
- changes to Chapter 2 (upto revision 1.23)
  - rewritten to reflect developments since publication of the MathML 1.0 Recommendation, for example XML, XSL, CSS and schemas
  - added reference to XML recommendation
  - removed error in description of allowed character in attribute values
- changes to Chapter 3 (upto revision 1.39)
  - the attribute `definitionURL` can have a URL or a URI as value
  - added sections about `menclose` and `meqno`
  - added attributes `beveled`, `numalign` and `denomalign` to `mfrac`, and updated text accordingly
  - made sure examples are correct, and fixed several typos
  - added sections on `mchar` and `mglyph`
  - adjusted description of `mstyle` and `mglyph`
  - added description of `mlabeledtr`
  - added examples for actuarial notation and long division
  - added width attribute to `mtable`
  - describe deprecated features
  - make use of `encoding` attribute more uniform
  - removed inferred `mtr` and `mtd`
- changes to Chapter 4 (upto revision 1.39)
  - discuss changed use of `apply`, and deprecation of `reln`
  - introduce `csymbol` and discuss the relation with `fn`
  - introduce the new category of elementary classifical functions
  - introduce new content elements `arg`, `real`, `imaginary`, `equivalent`, `approx`, `divergence`, `grad`, `curl`, `laplacian`, `size`, `vectorproduct`, `scalarproduct` and `outerproduct`
  - made sure examples are correct, and fixed several typos
  - the attribute `definitionURL` can have a URL or a URI as value
  - revised some of the default renderings
  - described the use of presentation markup inside `cn`
  - modified the example for root to indicate that the rendering with a radical sign is for integer degrees only
  - default rendering of `not` made to match example markup
  - added `minus` to the row for unary arithmetic in the table in section 4.2.3

  - make use of `encoding` attribute more uniform
  - changed description of the use of `bvar` in combination with `min` and `max`
  - describe deprecated features
  - make use of `encoding` attribute more uniform
- changes to Chapter 5 (upto revision 1.21)
  - added description of content-faithful transformation
  - updated to use `csymbol` and not `fn` in examples
  - define list of content that can appear in presentation
  - add attribute `xref` for cross-referencing purposes
  - added brief description of the elements `OMA`, `OMS` and `OMV`
  - added examples using XLink and namespaces
  - make use of `encoding` attribute more uniform
  - miscellaneous typographical corrections
- changes to Chapter 6 (upto revision 1.9) chap6 modifications, linking to tables
  - none
- changes to Chapter 7 (upto revision 1.25)
  - rewrote introductory text in section 7.2 and all text of section 7.2.1
  - rewrote many statements in future tense to present or past tense
  - reworked the text in acknowledgement of the fact that the top-level and interface elements for MathML are now in practice the same
  - rewrote the text about linking in accordance with the new XLink draft
  - revisited the material about interactions with embedded renderers to reflect the current state of DOM implementation
  - made sure examples are correct, and fixed several typos
  - describe deprecated features
  - make use of `encoding` attribute more uniform
  - rewrote text extensively to describe namespaces and CSS behaviors
- changes to Chapter 8 (upto revision 1.5)
  - this is a completely new chapter
  - moved IDL definitions to a new, non-normative appendix
- changes to Appendix A (upto revision 1.15)
  - renamed attribute `occurence` to `occurrence`
  - added global attribute `xref`
  - add links to tables for each entity set
- changes to Appendix B (upto revision 1.7)
  - none
- changes to Appendix C (upto revision 1.15)
  - completely rewritten
- changes to Appendix D (upto revision 1.14)
  - entries in operator dictionary are parametrized
  - operator dictionary has become non-normative part of the specification
  - new entries were added to operator dictionary
- changes to Appendix E (upto revision 1.19)
  - this is a completely new appendix, containing the IDL definitions that used to be in chapter 8
  - several interfaces were changed
  - add `xmlns` attribute declarations to `none`, `sep` and `mprescripts`
  - added various attributes and methods to reflect changes in the spec
  - replaced interface MathMLCollection with MathMLDOMImplementation
  - extended list of elements supported by interface MathMLpredefinedSymbol

  - reconcile various inconsistent uses of methods
- changes to Appendix F (upto revision 1.14)
  - added entries for XSL, XSLT and XSL FO
- changes to Appendix G (upto revision 1.11)
  - all members of first and second Math working group are listed
  - new addresses for Maple
  - removed ‘Publishers’ from affiliation of NP
- changes to Appendix H (upto revision 1.14)
  - completely new appendix, based on the logs obtained from CVS
- changes to Appendix I (upto revision 1.11)
  - added entry for XML Recommendation
  - added documents about XML Schemas
  - added entry for other W3C documents
  - changed first author of reference 5 to ‘Chaundy’
  - added revised edition of Ellen Swanson’s book
- general changes
  - text of specification now in XML form, with HTML and XHTML rendering by means of XSLT, and PDF rendering by means of XSLT and TeX
  - fixed errors in spelling and notation
  - normative examples of formulae are images, with a LaTeX equivalent
  - non-normative examples of formulae are HTML constructions wherever possible
  - improved cross-referencing

Appendix I

References (Non-Normative)

[Bray1998] Bray, Tim, Jean Paoli and C.M. Sperberg-Mcqueen; *Extensible Markup Language 1.0*, 10 February 1998, http://www.w3.org/TR/1998/REC-xml-19980210.

[Buswell1996] Buswell, S., Healey, E.R. Pike, and M. Pike; *SGML and the Semantic Representation of Mathematics*, UIUC Digital Library Initiative SGML Mathematics Workshop, May 1996 and SGML Europe 96 Conference, Munich 1996.

[Cajori1928] Cajori, Florian; *A History of Mathematical Notations*, vol. I & II. Open Court Publishing Co., La Salle Illinois, 1928 & 1929 republished Dover Publications Inc., New York, 1993, xxviii+820 pp. ISBN 0-486-67766-4 (paperback).

[Carroll1871] Carroll, Lewis [Rev. C.L. Dodgson]; *Through the Looking Glass and What Alice Found There*, Macmillian & Co., 1871.

[Chaundy1954] Chaundy, T.W., P.R. Barrett, and C. Batey; *The Printing of Mathematics. Aids for authors and editors and rules for compositors and readers at the University Press, Oxford*, Oxford University Press, London, 1954, ix+105 pp.

[Drucker1997] Drucker, Peter; Forbes, 10 Mar 1997 [quoted by Gene Klotz].

[Higham1993] Higham, Nicholas J., *Handbook of writing for the mathematical sciences*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1993. xii+241 pp. ISBN: 0-89871-314-5.

[Knuth1986] Knuth, Donald E., *The TEXbook*. American Mathematical Society, Providence, RI and Addison-Wesley Publ. Co., Reading, MA, 1986, ix+483 pp. ISBN: 0-201-13448-9.

[LieBos1996] Lie, Håkon Wium and Bert Bos; Cascading Style Sheets, level 1, W3C Recommendation, 17 Dec 1996, http://www.w3.org/pub/WWW/TR/REC-CSS1.

[OpenMath1996] OpenMath Release 1, December 1996; www.openmath.org.

[Pierce1961] Pierce, John R.; *An Introduction to Information Theory. Symbols, Signals and Noise.*, Revised edition of *Symbols, Signals and Noise: the Nature and Process of Communication* (1961). Dover Publications Inc., New York, 1980, xii+305 pp. ISBN 0-486-24061-4.

[Poppelier1992] Poppelier, N.A.F.M., E. van Herwijnen, and C.A. Rowley; *Standard DTD's and Scientific Publishing*, EPSIG News 5 (1992) #3, September 1992, 10-19.

[HTML4.0] Raggett, Dave, Arnaud Le Hors and Ian Jacobs; HTML 4.0 Specification, 18 Dec 1997, http://www.w3.org/TR/REC-html40/; section on data types.

[Spivak1986] Spivak, M. D. *The Joy of TEX A gourmet guide to typesetting with the AMS-TEX macro package*. American Mathematical Society, Providence, RI, MA 1986, xviii+290 pp. ISBN: 0-8218-2999-8.

[Swanson1979] Swanson, Ellen, *Mathematics into type. Copy editing and proofreading of mathematics for editorial assistants and authors*. Revised edition. American Mathematical Society, Providence, R.I., 1979. x+90 pp. ISBN: 0-8218-0053-1.

[Swanson1999] Swanson, Ellen, *Mathematics into type: Updated Edition*. American Mathematical Society, Providence, R.I., 1999. 102 pp. ISBN: 0-8218-1961-5.

[XLink] DeRose, Steve, David Orchard and Ben Trafford (editors), *XML Linking Language (XLink)*. World-Wide Web Consortium working draft, July 1999. (http://www.w3.org/TR/xlink)

[XPointer] Steve DeRose, Ron Daniel, Eve Maler (editors), *XML Pointer Language (XPointer)*. World-Wide Web Consortium working draft, Decemeber 1999. (http://www.w3.org/TR/xptr)

[XSLT] Clark, James (editor) *XSL Transformations (XSLT)*, version 1.0. World-Wide Web Consortium, October 1999. (http://www.w3.org/TR/xslt)

[Namespaces] Tim Bray, Dave Hollander, Andrew Layman (editors) *Namespaces in XML* World-Wide Web Consortium, January 1999. (http://www.w3.org/TR/REC-xml-names/)

[Behaviors] Vidur Apparao, Daniel Glazman, Chris Wilson (editors) *Behavioral Extensions to CSS* World-Wide Web Consortium, September 1999. (http://www.w3.org/TR/becss)

[DOM] Lauren Wood, Arnaud Le Hors, Vidur Apparao, Laurence Cable, Mike Champion, Joe Kesselman, Philippe Le H?ret, Tom Pixley, Jonathan Robie, Peter Sharpe, Chris Wilson (editors) *Document Object Model (DOM) Level 2 Specification* World-Wide Web Consortium, December 1999. (http://www.w3.org/TR/DOM-Level-2/)

[XHTML1.0] Steve Pemberton et al. (authors) *XHTML[tm] 1.0: The Extensible HyperText Markup Language* World-Wide Web Consortium, January 2000. (http://www.w3.org/TR/xhtml1/)

[XHTML1.1] Murray Altheim, Shane McCarron (editors) *XHTML[tm] 1.1: Module-based XHTML* World-Wide Web Consortium, January 2000. (http://www.w3.org/TR/xhtml11/)

[Modularization] Murray Altheim, Frank Boumphrey, Sam Dooley, Shane McCarron, Ted Wugofski (editors) *Modularization of XHTML* World-Wide Web Consortium, January 2000. (http://www.w3.org/TR/xhtml-modularization/)

[Building] Murray Altheim, Shane McCarron (editors) *Building XHTML Modules* World-Wide Web Consortium, January 2000. (http://www.w3.org/TR/xhtml-building/)

[XMLSchemas] Ashok Malhotra and Murray Maloney(authors) *XML Schema Requirements, W3C Note 15 February 1999* World-Wide Web Consortium, February 1999. (http://www.w3.org/TR/NOTE-xml-schema-req/)