Conversion of MathML to SVG via XSLT: pMML2SVG

Developer Documentation

Jérôme Joslet, Université de Liège `<jerome.joslet@student.ulg.ac.be>`
Justus H Piater
Professor
Université de Liège Faculty of Applied Sciences Department of Electrical Engineering and Computer Science

Conversion of MathML to SVG via XSLT: pMML2SVG: Developer Documentation

by Jérôme Joslet
Justus H Piater
Professor
Université de Liège Faculty of Applied Sciences Department of Electrical Engineering and Computer Science

Table of Contents

List of Figures

List of Examples

Main stylesheet

Main idea

Each MathML element can be viewed as a box that will be placed on the final SVG document. A box is represented by a minimum of six attributes that give information about its position and its size.

Attributes of a box

`X, Y`	Represent the two dimension coordinates of the upper left corner of the box.
`WIDTH, HEIGHT`	Represent the size of the box.
`BASELINE`	Represents the line on which the character will be aligned. Analogies can be made with the light guide lines on a lined sheet.
`FONTSIZE`	Determines the font size used in this box.

pMML2SVG works with the XML tree and transforms MathML to SVG in two passes. The first pass, called `formatting` mode, anotates each node of the MathML tree with information about position and size in order to compute a box. These annotations are placed as attributes on the node and belong to a temporary namespace named `t`. A namespace is a family of XML tags and attributes defined in an XML schemas. The second pass, named `drawing` mode, interprets annotations in order to draw the boxes on the SVG result canevas.

Some boxes need additionnal information to render correctly. For example, for the fraction, coordinates have to be added to place the fraction bar. Each element will describe which information is added to the tree and how it is handled.

An XSLT template is written for each MathML element and for each pass. It means that to implement a MathML element, two templates have to be written. One for the `formatting` mode and one for the `drawing` mode.

Name

svgMasterUnit

Synopsis

```
<xsl:param name="svgMasterUnit" select="'px'"/>
```

Name

initSize

Synopsis

```
<xsl:param name="initSize" select="50"/>
```

Description

This value cannot be changed by any MathML element. It can only be configured by setting it with the XSLT processor. This value can also be set by an external stylesheet that calls a MathML to SVG transformation. For example, the stylesheet that transforms the equation into picture in the XSL-FO code will set this value with respect to the current context.

Name

minSize

Synopsis

```
<xsl:param name="minSize" select="8"/>
```

Description

This parameter can be set by a `mstyle` element but it is not yet supported.

Name

delimPart

Synopsis

```
<xsl:variable name="delimPart"/>
```

Description

The structure is composed of `parts` tags that represent a horizontal or a vertical operator that have to be stretched. A part contains two to four `part` children that represent a glyph which composes the operator. The `parts` element can also have attributes. Here is a description of possible attributes for this tag:

`vname` Indicates which operator is stretched **vertically** using these glyphes to compose it.

`hname` Indicates which operator is stretched **horizontally** using these glyphes to compose it.

`hrotate` Indicates that the glyphes have to be rotated to compose the horizontal operator. For example, the over or under bracket is stretched using the same vertical glyphes than a normal vertical bracket. Therefore, the glyphes have to be rotated to become horizontal. This way of composing an operator is due to the unicode encoding that does not contain the horizontal glyphes to compose an over or under bracket.

`extenser` Determines towards which side the extenser has to be added. This attribute is used when the symbol is composed by only two glyphes. For example, the right floor operator is composed by a bottom part and an extenser, a right simple arrow is composed by a right arrow header and an extenser. In the case of the floor operator, the extenser must be added at the `top` of the other part, and for the arrow, the extenser is added on the `left` of the arrow head. This attribute can take four values:

- `top` The extenser will be added at the `top` of the other part.
- `bottom` The extenser will be added at the `bottom` of the other part.
- `left` The extenser will be added on the `left` of the other part.
- `right` The extenser will be added on the `right` of the other part.

The last `part` element is always the extenser, the other parts depend on the number of part element. When there are four `part` elements, the first element is the top or the left part, the second is the bottom or the right and the third is the middle. When there are three `part` elements, the first element is the top or the left part, and the second is the bottom or the right. When there are two `part` elements, the first part depends on the `extenser` attribute. If extenser is `top`, the first element is the bottom, if `extenser` is `bottom`, the first part is the top. If `extenser` is `left`, the first element is the right part and for `right` it is the left part.

Here are some examples of operators that have to be composed and the corresponding `parts` elements in the structure:

Example 1. Bracket parts

```
<parts vname="(" hname="&#65077;" hrotate="true">
  <part>&#9115;</part>
  <part>&#9117;</part>
  <part>&#9116;</part>
</parts>
<parts vname=")" hname="&#65078;" hrotate="true">
  <part>&#9118;</part>
  <part>&#9120;</part>
  <part>&#9119;</part>
</parts>
```

Example 2. Floor parts

```
<parts vname="&#8970;" extenser="top">
  <part>&#9123;</part>
  <part>&#9122;</part>
</parts>
<parts vname="&#8971;" extenser="top">
  <part>&#9126;</part>
  <part>&#9125;</part>
</parts>
```

Example 3. Arrow parts

```
<parts hname="&#8594;" extenser="left">
  <part>&#8594;</part>
  <part>&#9135;</part>
</parts>
<parts hname="&#8596;">
  <part>&#8594;</part>
  <part>&#8592;</part>
  <part>&#9135;</part>
</parts>
```

Example 4. Curly Bracket parts

```
<parts vname="{{" hname="&#65079;" hrotate="true">
  <part>&#9127;</part>
  <part>&#9129;</part>
  <part>&#9128;</part>
  <part>&#9130;</part>
</parts>
```

Name

delimScale

Synopsis

```
<xsl:variable name="delimScale" select="('|', '/', '\', '#',         '#', '#', '#'
```

Description

It contains an XSLT sequence of characters. These characters will be stretched using the `scale` SVG transformation.

Name

thin

Synopsis

```
<xsl:variable name="thin" select="'0.0625em'"/>
```

Name

medium

Synopsis

```
<xsl:variable name="medium" select="'0.1875em'"/>
```

Name

thick

Synopsis

```
<xsl:variable name="thick" select="'0.3125em'"/>
```

Name

rowElement

Synopsis

```
<xsl:variable name="rowElement" select="('mrow', 'mtd', 'msqrt', 'mstyle', 'merror
```

Description

It contains an XSLT sequence of string that represent a list of MathML elements.

Name

getMiddle — Determines the space between the baseline and the middle of the line. This middle is the horizontal bar of the plus operator.

Synopsis

```
<xsl:function name="func:getMiddle">
<xsl:param name="fonts"/>
<xsl:param name="variant"/>
  ...
</xsl:function>
```

Description

This value is determined by computing the top edge Y coordinate of the `-` operator by using the `findHeight` function.

See

`findHeight`

Parameters

fonts	Current font list.
variant	Variant for the fonts, this variant can be `-Italic`, `-Bold`, `-Bold-Italic` or empty.

Returns

Returns the space between the baseline and the middle of the line.

Name

computeSize

Synopsis

```
<xsl:template name="computeSize">
<xsl:param name="initSize" tunnel="yes"/>
<xsl:param name="sizeMult" tunnel="yes"/>
<xsl:param name="minSize" tunnel="yes"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
  ...
</xsl:template>
```

Description

The font size is computed by using the `computeSizeMult` function that returns a multiplication factor with respect to the current `scriptLevel` and `sizeMult`. The initial font size is divided by this factor if the current `scriptLevel` is lower than 0, and is multiplied by it if the current `scriptLevel` is greater than 0.

The size is then compared to `minSize` to return this new font size or the minmum font size. This comparison is done to avoid getting a font size that is too small in order to display correctly an expression.

See

`computeSizeMult`

Parameters

Parameters from tunnel	All the function paramaters are retrieved from the tunnel. These parameters are described in detail in the root element description.

Returns

Returns the current font size.

Name

computeSizeMult — Compute the factor that will multiply (or divide) the initial size in the computeSize function.

Synopsis

```
<xsl:function name="func:computeSizeMult" as="xs:double+">
<xsl:param name="sizeMult"/>
<xsl:param name="scriptlevel"/>
  ...
</xsl:function>
```

Description

This recursive function compute `sizeMult` exponent `scriptlevel`. It is done recursively by multiplying `sizeMult` by the result of the recursion. The `scriptlevel` is decremented by one at each recursion call. The basic case is when this value falls to zero and the function simply returns 1.

Note that this function is never recusively called with the same paramaters as in the first call. It is impossible since, the `scriptlevel` is always decremented by one.

Parameters

sizeMult	Size multiplier represents the factor by wich the initial size has to be multiplied when the script level changes.
scriptlevel	Current value for the `scriptlevel`. At each recursion, this value is decremented by one to compute the final factor.

Returns

Returns the factor.

Name

unitInPx

Synopsis

```
<xsl:template name="unitInPx" as="xs:double+">
<xsl:param name="valueUnit"/>
<xsl:param name="fontSize"/>
<xsl:param name="default" select="0"/>
  ...
</xsl:template>
```

Description

This template simply applies a computation with respect to the value unit. It can handles all these units: literal, px, em, ex, % and no unit. A space literal is computed by using the `getSpaceLiteral` template. For example, by using this template, 3em will be computed 3 * `fontSize`, 5 will be computed 5 * `default`, etc.

See

`getSpaceLiteral`

Parameters

valueUnit	The original measure to handle.
fontSize	The current font size, this paramater is used to compute relative unit value.
default	This parameter is used to compute percentage or no unit measure.

Returns

Returns the value expressed in the master unit.

Name

getSpaceLiteral

Synopsis

```
<xsl:template name="getSpaceLiteral">
<xsl:param name="literal"/>
<xsl:param name="veryverythinmathspace" tunnel="yes"/>
<xsl:param name="verythinmathspace" tunnel="yes"/>
<xsl:param name="thinmathspace" tunnel="yes"/>
<xsl:param name="mediummathspace" tunnel="yes"/>
<xsl:param name="thickmathspace" tunnel="yes"/>
<xsl:param name="verythickmathspace" tunnel="yes"/>
<xsl:param name="veryverythickmathspace" tunnel="yes"/>
  ...
</xsl:template>
```

Description

This template simply browses all possible literal names and returns the corresponding space value. It supports the following literals: `veryverythinmathspace`, `verythinmathspace`, `thinmathspace`, `mediummathspace`, `thickmathspace`, `verythickmathspace`, `veryverythickmathspace`.

Parameters

literal	The literal name.
*mathspace	Represents the value of the space literals. These values are retrieved from tunnel since the `mstyle` element enables to modify them.

Returns

Returns the value expressed by a space literal.

Name

chooseAttribute — Selects the best value between the three paramaters.

Synopsis

```
<xsl:function name="func:chooseAttribute">
<xsl:param name="user"/>
<xsl:param name="herited"/>
<xsl:param name="default"/>
  ...
</xsl:function>
```

Description

This function is used to retrieve style attributes. The value specified by users has the priority and will be chosen if it is not empty. The second choice is the herited value from a parent if it is not empty. And, finally, the default one is choosen if all other values are empty.

Parameters

user	The value wanted by users. Specified via an attribute to an element.
herited	It is the value herited from a parent.
default	It is the default value from the specification.

Returns

Returns the choosen attribute.

Name

getFontNameVariant — Computes the variant of the font name with respect to the style parameters.

Synopsis

```
<xsl:function name="func:getFontNameVariant">
<xsl:param name="mathvariant"/>
  ...
</xsl:function>
```

Description

This function computes the variant from the `mathvariant` style attribute. It simply checks if this attribute contains the string `bold` and the string `italic`. In the future, more styles have to be implemented.

Parameters

mathvariant Value of the `mathvariant` attribute from an element.

Returns

Returns the variant: `-Bold`, `-Italic` or `-Bold-Italic`.

Name

setStyle — Computes a CSS style rule for an element with respect to all style attributes.

Synopsis

```
<xsl:function name="func:setStyle">
<xsl:param name="mathvariant"/>
<xsl:param name="mathcolor"/>
<xsl:param name="mathbackground"/>
  ...
</xsl:function>
```

Description

This function computes the CSS style attribute. It checks if the `mathvariant` contains the string `bold` and the string `italic` and adds the correct CSS rules with respect to this verification. It also adds a `fill` rule to change the color of the drawed element width respect to the `mathcolor` pamareter.

Parameters

mathvariant	Value of the `mathvariant` attribute from an element.
mathcolor	Value of the `mathcolor` attribute from an element.
mathbackground	Value of the `mathbackground` attribute from an element.

Returns

Returns the CSS style attribute for an element.

Name

stringWidth — Function that simply calls the stringWidth template.

Synopsis

```
<xsl:function name="func:stringWidth">
<xsl:param name="str"/>
<xsl:param name="fontName"/>
<xsl:param name="variant"/>
  ...
</xsl:function>
```

Description

This function was created because, in some cases, it is more simple to call a function than a template. The `stringWidth` template computes the width of a string with respect to the metrics files.

See

stringWidth
template.

Parameters

str	String to compute the width.
fontName	List of fonts that will be used to render the string.
variant	Variant for the font.

Returns

Returns the value of the `stringWidth` template.

Name

stringWidth

Synopsis

```
<xsl:template name="stringWidth">
<xsl:param name="str"/>
<xsl:param name="strLen"/>
<xsl:param name="i" select="1"/>
<xsl:param name="size" select="0"/>
<xsl:param name="fontName" select="'STIXGeneral'"/>
<xsl:param name="variant" select="''"/>
  ...
</xsl:template>
```

Description

It is a recursive function that sums the width of each character that composes the string. A correction is added to the top and to the bottom of the string. This correction is computed using the bounding box of, respectively, the first and the last character of the string. These two corrections are compute in the `leftBearing` and `rightBearing` variables.

Note that the recursion will always end because the index `i` will finally reached `strLen`. Moreover, it is never recursively called with the same parameter values as the first call because the index `i` is always incremented by one.

See

`stringWidth` template.

Parameters

str	String to compute the width.
strLen	Number of character in the string.
i	Index of the character that is currently handled.
size	This parameter is an accumulator that contains the size (in em) for the first `i-1` characters of the string.
fontName	List of fonts that will be used to render the string.
variant	Variant for the font.

Returns

Returns the width of the string in em.

Name

math:math — Root element of the transformation.

Synopsis

```
<xsl:template match="math:math">
<xsl:param name="svgMasterUnit" select="$svgMasterUnit" tunnel="yes"/>
<xsl:param name="initSize" select="$initSize" tunnel="yes"/>
<xsl:param name="sizeMult" select="0.71" tunnel="yes"/>
<xsl:param name="minSize" select="$minSize" tunnel="yes"/>
<xsl:param name="svgBorder" select="$initSize div 5" tunnel="yes"/>
<xsl:param name="errorMargin" select="$initSize div 10" tunnel="yes"/>
<xsl:param name="rightSwitch" select="$initSize div 15" tunnel="yes"/>
<xsl:param name="numDenSpace" select="$initSize div 5" tunnel="yes"/>
<xsl:param name="overUnderSpace" select="$initSize div 10" tunnel="yes"/>
<xsl:param name="tableSpace" select="$initSize div 2" tunnel="yes"/>
<xsl:param name="fracWidMarg" select="$initSize div 15" tunnel="yes"/>
<xsl:param name="rtTopSpc" select="$initSize div 6" tunnel="yes"/>
<xsl:param name="rtFrnSpcFac" select="0.5" tunnel="yes"/>
<xsl:param name="fontName" select="$fontName" tunnel="yes"/>
<xsl:param name="scriptlevel" select="0" tunnel="yes"/>
<xsl:param name="displayStyle" select="'true'" tunnel="yes"/>
<xsl:param name="veryverythinmathspace" select="'0.055556em'" tunnel="yes"/>
<xsl:param name="verythinmathspace" select="'0.111111em'" tunnel="yes"/>
<xsl:param name="thinmathspace" select="'0.166667em'" tunnel="yes"/>
<xsl:param name="mediummathspace" select="'0.222222em'" tunnel="yes"/>
<xsl:param name="thickmathspace" select="'0.277778em'" tunnel="yes"/>
<xsl:param name="verythickmathspace" select="'0.333333em'" tunnel="yes"/>
<xsl:param name="veryverythickmathspace" select="'0.388889em'" tunnel="yes"/>
  ...
</xsl:template>
```

Description

The root element is the starting point of the transformation. This template is called when a `math` element is found in the document that is currently transformed. This template will call the two passes of the transformation. First, it will retrieve the annotated tree from the `formatting` mode by applying `formatting` mode template on the entire MathML tree. Then, it will retrieve the total width and height of the expression and writes the header of the SVG file. It will also write metadata information about the baseline. This information is used to shift the SVG picture when pMML2SVG is calling from an other stylesheet and when picture must be embedded into a text line. Finally, the root element will call the `drawing` mode on the annotated tree to draw all element on the canevas.

All elements follow the same scheme. In `formatting` mode, the font size is first computed, all attribute for the element are retrieved. After that, the children elements are computed if necessary, then the box is created by computing all its attribute and finally the tree node is annotated.

In `drawing` mode, the `X` and `Y` coordinates of the box is first computed. Then, the children are drawn if necessary (by calling their `drawing` mode) and, finally the elements of the box itself are added on the canvas (fraction bar, boxes, etc.).

The first element that is called in each transformation is the `math` element. This element is the root of each MathML equation.

Each template in the `formatting` mode must take at least three parameters:

`X, Y` Represent the initial upper left corner of the box where the element will be drawn. If the baseline is not set, `Y` value is used to set a new baseline for the current element.

`BASELINE` By default, this value is zero. It means that no baseline has been created and that the current element will decide where its baseline will be. In that case, the element will align its top edge on the initial `Y` value. If this parameter is set, the element has to be aligned on this baseline.

In the `drawing` mode, at least two paramaters are required:

`xShift, yShift` These values determine if the element has to be shifted to be correctly displayed. For example, when rendering a fraction. Both numerator and denominator will be aligned on the baseline by the formatting mode. Therefore, numerator has to be shifted to the top and denominator to the bottom to find their final correct place. When drawing, numerator and denominator elements will receive shift values via these paramaters.

Parameters

Parameters can be retrieved through the tunnel: global parameters or style parameters. A tunnel is a way to forward parameters to all elements through the XML tree without sending them explicitly. It means that each template implicitly forwards these paramaters to all templates they call. Global parameters are set by default by the root template. All these values can be changed if the stylesheet is called via another stylesheet. Some of them can also be changed by setting parameters when executing the transformation with an XSLT processor. Here is a description of all global paramaters.

Description of global parameters

`svgMasterUnit` Determines the default unit that will be used to render the SVG picture. The default unit is pixel (px).

`initSize` Determines the initial font size. This value cannot be changed by any MathML element. It can only be configured by setting it with the XSLT processor. This value can also be set by an external stylesheet that calls a MathML to SVG transformation. For example, the stylesheet that transforms the equation into picture in the XSL-FO code will set this value with respect to the current context. By default, the value of this parameter is 50.

`sizeMult` `sizeMult` is a factor by which the font size has to be multiplied to render script element. This parameter works with `scriptLevel`. For example, if the script level is 5, you have to multiply the initial font size by `sizeMult` 5 times. This parameter can be set by a `mstyle` element. The default size multiplier is 0.71.

`scriptlevel` Determines the number of times you will have to multiply the initial font size by `sizeMult` to render the current element. This parameter can be set by a `mstyle` element. The default script level is 0.

`displayStyle` Determines the display scheme on some elements, for example, if display style is `false`, `mover`, `munder` and `munderover` limit of summation or integral operators will be moved from top and bottom to right. This parameter can be set by a `mstyle` element. The default script level is `true`.

`minSize` Determines the minimal font size. This parameter can be set by a `mstyle` element but it is not yet supported. By default, this value is 8.

`svgBorder` Determines the size of the transparent border that surrounds the picture. By default, this value depends on `initSize` and is `initSize` div 5. This value will also set the initial `X` and `Y` coordinates to launch the transformation.

`rightSwitch` Determines the size of the space between a base and its subscript or superscript. This parameter is used in `msub`, `msup` and `msubsup` elements. By default, this value depends on `initSize` and is `initSize` div 15.

`numDenSpace` Determines the space between the numerator and the denominator. This parameter is used in `mfrac` element. By default, this value depends on `initSize` and is `initSize` div 5.

`overUnderSpace` Determines the size of the space between a base and its overscript or underscript. This parameter is used in `munder`, `mover` and `munderover` elements. By default, this value depends on `initSize` and is `initSize` div 10.

`tableSpace` Determines the size of the space between two cells of a table. This parameter is used in `mtable`, `mtd` and `mtr` elements. By default, this value depends on `initSize` and is `initSize` div 2.

`fracWidMarg` Determines the size of the fraction bar that outpasses the numerator or the denominator. This parameter is used in `mfrac` element. By default, this value depends on `initSize` and is `initSize` div 15.

`rtTopSpc` Determines the size of the space over the base of a root to render the radical line. This parameter is used in `msqrt` and `mroot` elements. By default, this value depends on `initSize` and is `initSize` div 6.

`rtFrnSpcFac` Determines the size of the space before the base of a root to render the radical line. By default this value is 0.5.

`fontName` Determines the fonts that will be used to render elements. This parameter is a list of fonts separated by a comma. By default, the font list is `STIXGeneral,STIXSize1`. The way you can change this parameter is explained in the user guide.

`Math space parameters` These parameters are used to determine the value of a space literal. The space literal can be used when a MathML attribute requires a horizontal measure. The default value for these parameters comes from the MathML specification. These values can be changed with the `mstyle` element.

Style tunnel parameters are used to implement the heritage of style. Currently, only `mathvariant`, `mathcolor` and `mathbackground` are implemented. Other style attributes are easy to add following the current scheme.

Description of each style paramaters

`mathvariant` This attribute is partially supported and enables users to put the style in bold, italic or both. It also enables to change the font used to render an element. This last functionnality is not yet supported.

`mathcolor` Enables users to change the color of an element. This attribute is fully supported.

`mathbackground` Enables users to change the background color of an element. Currently, this attribute does nothing because SVG element does not have background to color. To fully implement this attribute, we have to draw a colored rectangle that has the size of the element.

All these tunnel parameters are forwarded to both formatting and drawing mode. They are available everywhere and can be modified by all elements. However, this modification is only reflected on the children, it is a good way to implement the inherited style attributes.

Formatting mode

Name

math:mi|math:mn|math:mtext|math:ms (in formatting mode) — Formatting a token element.

Synopsis

```
<xsl:template match="math:mi|math:mn|math:mtext|math:ms" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="fontName" tunnel="yes"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="mathvariant" tunnel="yes"/>
<xsl:param name="mathcolor" tunnel="yes"/>
<xsl:param name="mathbackground" tunnel="yes"/>
  ...
</xsl:template>
```

Description

All these elements are treated the same way with a few exceptions. Therefore, in the implemented stylesheet, they share the same template, both for formatting and for drawing. These elements are the leaf of the MathML tree, they do not have any children, they only contain text.

After computing the font size of these elements, the `ms` `lquote` and `rquote` attributes are retrieved. These attributes determine which symbol will be used to surround the text, respectively, on the left and on the right. After that, the text content of the element is retrieved. `lquote` is added before the first character and `rquote` after the last one if the element is `ms`.

Next, the font variant is computed to retrieve the width and the height of the text. An `mi` element with one letter (except infinity symbol) has to be displayed in italic, so the font variant is computed using that particularities. All the parameters of the box can now be computed: the height is given by font metrics file, the width is computed using the `stringWidth` function and the baseline is set on the bottom of the text with descender stretching under it. The box also contains an other measure: `HEIGHTOVERBASELINE` which is the height of the box from its baseline to its top edge. Here is a figure that represent the box for a token element:

Figure 1. Box for a token element

The left bearing of the box is computed to shift the character inside the box. If no bearing is computed, some characters are drawn outside the box. For exemple, the left part of an italic **f** goes out of the left side of the box if no shift value is added by using the left bearing. The right bearing is computed to place the subscript closer to some characters. In the case of an italic **f**, if the subscript is placed after the letter box, it will be too far away from **f**. If the right bearing is withdrawn from the coordinates of the right side of the box, the subscript will be drawn closer.

Finally, the tree node is annotated with the box representation and with style (attributes `STYLE`) information about the box. A shift value (`SHIFTX`) is also added when the token has a left bearing. The left

bearing is a negative value from the left value of the first character bounding box. It occurs, for example, with an italic **f**.

Name

math:mspace (in formatting mode) — Formatting a space.

Synopsis

```
<xsl:template match="math:mspace" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
  ...
</xsl:template>
```

Description

`mspace` follows the general scheme. After computing the font size, all attributes are retrieved and computed in pixel. The box representation is computed and the tree is annotated with these values.

Parameters

This element, that represents a space, has three attributes that determine its size:

`width` Determines the width of the space.

`height` Determines the size of the box over the baseline.

`depth` Determines the size of the box under the baseline.

Name

math:mo[not(@t:stretchVertical) or @t:stretchVertical != true()] (in formatting mode) — Formatting an operator.

Synopsis

```
<xsl:template match="math:mo[not(@t:stretchVertical) or @t:stretchVertical != true
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="fontName" tunnel="yes"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
<xsl:param name="mathvariant" tunnel="yes"/>
<xsl:param name="mathcolor" tunnel="yes"/>
<xsl:param name="mathbackground" tunnel="yes"/>
<xsl:param name="thickmathspace" tunnel="yes"/>
  ...
</xsl:template>
```

See

Variables `delimPart` and `delimScale`. Function `chooseEntry`.

Description

It is one of the most complex elements to render. It has a lot of attributes that determine many different ways to display it. The default behaviour of operators is contained in a dictionary called `operator dictionary`. This dictionary, coming from the specification [1] , is implemented in the file `operator-dictionary.xml`. It has been implemented in XML to make access easier using XPath. Modifications have been done to add useful information for our renderer and to add new characters. Characters `Prime` and `Times` have been added to the dictionary to facilitate the rendering of these elements. New attributes have also been added:

`stretchHorizontal` If an operator has to stretch, this attributes tells our renderer that it will stretch horizontally.

`stretchVertical` If an operator has to stretch, this attributes tells our renderer that it will stretch vertically.

These two attributes can be both set to true. In this case, the operator has to stretch vertically and horizontally. None of these operators will be stretched in the current version of pMML2SVG.

The formatting mode for a `mo` element has two different behaviours. The first one is the normal mode that annotate the tree like the other elements. The second is used to correct the annotation of the tree when the operator has to be stretched vertically.

The specification tells us that such an operator should have the size of the biggest non-stretchy element present in the same row of it. Therefore, when an operator has to be stretched, the bottom and the top `Y` of this big element have to be known to compute the final size of the stretched operator. When the second mode is called, these two values are retrieved by the following template paramaters: `upperY` and

[1] `http://www.w3.org/TR/2003/REC-MathML2-20031021/appendixf.html`

`lowerY`. The way these values are computed and how this second template mode is called is explained in detail in the `alignChild` template.

This template is the normal mode, the correcting mode takes part in another template that is exmplained further.

First of all, all attribute values of the operator are retrieved. To determine the default behaviour of these values, the operator dictionary entries for this operator are retrieved. It is done by using XPath and the `document` function. This function is used to browse an external file. After that, the best operator dictionary entry is chosen with respect to the number of entries and the `form` attribute. If there is only one entry, this entry is chosen. If there is more than one entry, a default `form` attribute has to be computed. The rules to determine it are:

- If the operator is a member of a row, if there is more than one element in this row (excluding `mspace`) and if this operator is the first element in the row (excluding `mspace`), the `form` attribute is `prefix`
- If the operator is a member of a row, if there is more than one element in this row (excluding `mspace` and if this operator is the last element in the row (excluding `mspace`), the `form` attribute is `postfix`
- In all other cases, the `form` attribute is `infix`.

If there is an entry with this `form` attribute value, this entry will be chosen. If not, an entry will be chosen with preference to `infix` `form` attribute value, then `postfix` and finally `prefix`. This choosing rule is implemented in the function `chooseEntry`.

After choosing an entry, all other attributes will be finally retrieved. The value will be the user's specified one, if it exists, then, the value from the operator dictionary and finally a default value from the specification. The following attribute is retrieved:

`lspace, rspace`	Determine the space around the operator, respectively, on the left and on the right. Default value is `thickmathspace`.
`stretchy`	Determines if an operator has to be stretched. Default value is `false`. If this value is `true`, `stretchHorizontal` and `stretchVertical` variables are retrieved from the dictionary, if it is possible. In all other cases, these two last values are set to `false`. These variables are specific to pMML2SVG renderer.
`symmetric`	Determines if the operator will be stretched symmetrically. The default value is `true`.
`maxsize, minsize`	Determine, respectively, the maximum and minimum size of an operator. These two attributes are used to control the stretching of the operator. The default value is, respectively, `infinity` and `1`.
`largeop`	Determines if the operator is a large operator such as integral, summation, etc. The default value is `false`. If `displayStyle` values, from tunnel, and if `largeop` are `true`, then the operator will be rendered with higher font size. Typically, the `scriptlevel` to render this operator will decrease by one.
`movablelimits`	Determines if the limit under or above an operator (such as integral, summation, etc.) can be moved and be rendered on the left of the operator instead of under or above. The default value is `false`. This attribute is not yet used in pMML2SVG.
`accent`	Determines if an operator must behave like an accent. The default value is `false`. This attribute is used to correct the vertical position of accent operator such as circumflex accent, etc.

After all attributes have been retrieved, the font size for the box is computed. The font size has to be bigger if the `largeop` attribute is `true`. Therefore, the `scriptlevel` value is decremented by one in this case. Otherwise, the font size is computed normally.

Some operators have to be replaced by similar glyphes to be retrieved in the font metrics. It is the case with the under (and over) brackets.

After that, a correction is computed if the operator is an `accent`. This correction includes the computation of the height of each glyph part that compose the operator if this last has to be stretched. It is done by retrieving the bouding box of each part of the composed operator. This correction is necessary since the parts that compose an operator have a higher height than the non-composed operator.

The left and right bearings are also computed the same way as in other tokens. They have the same behaviour as in others tokens.

The box size and position is then computed and the computation in pixel of `minsize`, `maxsize`, `lspace` and `rspace` is done.

Finally, the tree node is annotated with box information, stretchy information (`STRETCHY`, `stretchHorizontal` and `stretchVertical` attributes), `minsize` and `maxsize` (in pixel) that will be used when the operator will be corrected to stretch, `lspace` and `rspace` (in pixel), `EMBELLISH` information, style information, `SYMMETRIC` information that will be used when the operator will be corrected to stretch and a shift value (`ACCENTSHIFT`) that is used to correct the vertical position of an accent.

The `EMBELLISH` information is used to know where to add `lspace` and `rspace`. The specification tells us what an embellished operator is:

Embellished operator definition

- An `mo` element.
- One of the elements `msub`, `msup`, `msubsup`, `munder`, `mover`, `munderover`, `mmultiscripts`, or `mfrac` whose first argument exists and is an embellished operator.
- A row whose arguments consist (in any order) of one embellished operator and zero or more space-like elements.

Adjustements have to be done when an embellished operator is computed. For example, if an `munder` element is an embellished operator, the space determined by `lspace` and `rspace` has to be placed around this `munder` element and not around its first `mo` child.

Name

math:mo[@t:stretchVertical = true()] (in formatting mode) — Correcting mode for an operator element.

Synopsis

```
<xsl:template match="math:mo[@t:stretchVertical = true()]" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="upperY" select="0" tunnel="yes"/>
<xsl:param name="lowerY" select="0" tunnel="yes"/>
<xsl:param name="fontName" tunnel="yes"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
  ...
</xsl:template>
```

Description

Some operators have to stretch symmetrically, it depends on the `symmetric` attribute value, it means than its size above the middle of the expression is equal to its size under this middle. The middle can be viewed as the position of the minus operator in the expression. Therefore, the first three lines of this mode (after size computation) are used to determine the height of the delimiter with respect to the biggest element and the `symmetric` attributes.

The next computation implements the behaviour of `minsize` and `maxsize` `mo` attributes. Since the correcting mode works directly on the annotated tree computed by the normal mode, these two attributes are retrieved directly from it.

Since the parts that compose an operator have a bigger width than the normal operator, the width of the box has to be corrected too. It is done by retrieving the bounding box of each part. These bounding boxes are used to compute the new width.

Finally, the box represention is corrected and directly annotated in the tree.

Name

chooseEntry — Chooses best entry in Operator Dictionary with respect to specification rules.

Synopsis

```
<xsl:function name="func:chooseEntry">
<xsl:param name="forms"/>
<xsl:param name="nodes"/>
  ...
</xsl:function>
```

Description

This function checks if the first form from the `forms` attribute exists in the operator dictionary `entries`. If not, the recursion is called with the next form entries.

Note thate recursion is never called with the same parameters as the first function call because an element is always removed from the `forms` sequence. Therefore, the recursion will always end because the size of `forms` sequence decrease and fall down to 0.

Parameters

forms — Sequence of `form` attribute string ordered by preference: user specified, rules from `form` attributes, `infix`, `postfix`, `prefix`.

nodes — Entries in the operator dictionary for the current operator.

Returns

Returns the best entry from the operator dictionary entries.

Name

math:math|math:mrow|math:merror|math:mphantom|math:menclose|math:mstyle (in formatting mode) — Formatting a box element.

Synopsis

```
<xsl:template match="math:math|math:mrow|math:merror|math:mphantom|math:menclose|m
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="errorMargin" tunnel="yes"/>
<xsl:param name="sizeMult" tunnel="yes"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
<xsl:param name="mathvariant" tunnel="yes"/>
<xsl:param name="mathcolor" tunnel="yes"/>
<xsl:param name="mathbackground" tunnel="yes"/>
<xsl:param name="veryverythinmathspace" tunnel="yes"/>
<xsl:param name="verythinmathspace" tunnel="yes"/>
<xsl:param name="thinmathspace" tunnel="yes"/>
<xsl:param name="mediummathspace" tunnel="yes"/>
<xsl:param name="thickmathspace" tunnel="yes"/>
<xsl:param name="verythickmathspace" tunnel="yes"/>
<xsl:param name="veryverythickmathspace" tunnel="yes"/>
  ...
</xsl:template>
```

See

`subMrow`

Description

All these elements are considered to have the same grouping comportment. Therefore, they are all handled the same way in the same template with some exception in the code.

First of all, we retrieve the number of children of the element. If this number is zero, an empty box is created and the tree is annotated with that box. If the number of children is greater than zero, the element will be treated normally. This distinction is used to handle correctly empty `mrow` that is used frequently.

As usual, all attributes are first retrieved. The `notations` `menclose` attribute is retrieved and all multiple spaces are replaced by one space. This attribute is used to determine which element(s) will enclose the row. It can contain more than one notation. For example, a row can be enclosed by both a circle and a box. After that, common style attributes are retrieved (there are not all implemented yet) and finally `mstyle` attributes are retrieved (in reality, only `scriptlevel` is retrieved here because it needs more complex treatment than others). The currently supported attributes are:

`scriptlevel`	Modifies the current level of the font size.
`displaystyle`	Modifies the rendering of some elements.
`Space literals (medium-mathspace, etc.)`	Modify the size value of space literals.

`scriptsizemultiplier`	Modifies the `sizeMult` factor.

`mstyle` attributes are quite different from other attributes because they have to be transmitted to their children. Therefore, they are retrieved when the child templates are called and only when the current element is an `mstyle` tag. Before formatting all children, new values for `X` and `Y` are computed. These new values will help to add more spaces around the children because `menclose` and `merror` need them to add elements (boxes, circle, root sign, etc.).

After that, children are computed using a template that will align them on the same baseline: `subMrow`. It takes four arguments: the new computed `X` and `Y` values, the baseline and all the child elements. This template will also correct the operator that has to stretch vertically by calling appropriate templates. After that, the highest right box side of children are retrieved to compute the width of the box. The height and `Y` information is computed by retrieving the lowest top side box and the highest bottom side box of children. A shift value is also computed if the children are getting out of the canvas. For example, if the baseline is at 20 and if a child has a height of 40, it will go out of the canvas by 20. Therefore, all the children have to be shifted to be drawn correctly on the canevas. After that, the baseline for this box is computed by using the shift value and the lowest baseline of all children.

Finally, the tree is annotated with box information, with the shift value, with value from its operator child (`EMBELLISH`, `LSPACE`, `RSPACE`, `stretchVertical` and `ACCENT`) if the row is considered as an embellished operator and `NOTATION` attribute for `menclose` element.

Name

subMrow

Synopsis

```
<xsl:template name="subMrow">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="nodes"/>
  ...
</xsl:template>
```

See

`alignChild`, `getStretchyEmbellished`

Description

This template is used to align the children of a row on the same baseline. It also calls stretchy correction on operators that must stretch vertically. The first part of this function is to compute all the children on the same baseline. To do that, it calls a template `alignChild` that takes the same parameters plus a `firstChild` parameter that is used to determine which element is the first child. This first child will give its basline attribute to all other children in order to align all children on the same baseline.

After that, the function corrects the elements that must stretch vertically. All the stretchy embellished operators are first retrieved by using `getStretchyEmbellished`. If there are no stretchy embellished operator, nothing is done and all the annotated children elements are returned. In the other case, a stretchy correction may be done.

If a stretchy correction has to be done, the lowest and highest `Y` of all non stretchy children have to be retrieved to know the final size of the stretchy operators. To retrieve these children, `getNonStretchyEmbellished` function is used. If there is no element that does not stretch, nothing is done and all the annotated children elements are returned without any correction. In the other cases, a stretchy correction is done.

Now that the non-stretchy elements are retrieved, the lowest and highest `Y` can be computed and the `alignChild` template is called again to recompute the row with these new parameters. All the elements have to be recomputed because if an operator has to be stretched, its width will be greater. Therefore, all the elements that follow it must have a new `X` coordinate. Finally, all elements are returned.

Parameters

x, y, baseline	Formatting mode required parameters.
nodes	Elements to handle in the template.

Returns

Returns all computed alements of the row.

Name

alignChild

Synopsis

```
<xsl:template name="alignChild">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="nodes"/>
<xsl:param name="firstNode" select="1"/>
  ...
</xsl:template>
```

Description

This template is called from `subMrow` and is used to compute and align a group of elements on the same baseline. Typically, these elements will be a part of a row. This template will simply browse all the elements, compute each of them by calling the appropriate template in formatting mode and give the baseline of the first non stretchy element to all other elements in order to align them on the same baseline. To know which element is the first non stretchy one, the `firstChild` parameter is used. The `X` coordinate value will be incremented by the size of the current element to compute the next one.

Spaces are sometimes added between two elements. Typically, it will be done between an `msub`, `msup` or `msubsup` element and an other element that is not an operator.

Parameters

x, y, baseline	Formatting mode required parameters.
nodes	Elements to handle in the template.
firstNode	Determines if the current element is the first one in the row. It will give its baseline to all other elements.

Returns

Returns all elements aligned on the same baseline.

Name

getStretchyEmbellished

Synopsis

```
<xsl:template name="getStretchyEmbellished">
<xsl:param name="nodes"/>
<xsl:param name="mode" select="'v'"/>
  ...
</xsl:template>
```

See

`isEmbellished`

Description

This function simply browses each node and checks if it is an embellished operator using `isEmbellished` function. If the function returns `true`, the node is copied. Otherwise, nothing is done.

Parameters

nodes Elements to check.

mode Stretching mode: `v` is to retrieve the operators that stretch vertically (default value), `h` is for horizontally, and `b` is for both vertically and horizontally.

Returns

Returns all the embellished operators that have to stretch vertically, horizontally or both.

Name

getNonStretchyEmbellished

Synopsis

```
<xsl:template name="getNonStretchyEmbellished">
<xsl:param name="nodes"/>
<xsl:param name="mode" select="'v'"/>
  ...
</xsl:template>
```

See

`isEmbellished`

Description

This function simply browses each node and checks if it is an embellished operator using `isEmbellished` function. If the function returns `true`, nothing is done. Otherwise, the node is copied.

Parameters

nodes Elements to check.

mode Stretching mode: `v` is to retrieve the operators that stretch vertically (default value), `h` is for horizontally, and `b` is for both vertically and horizontally.

Returns

Returns all the elements that are not an embellished operator that have to stretch vertically, horizontally or both.

Name

isEmbellished — Checks if an element is an embellished operator that has to stretch vertically, horizontally or both.

Synopsis

```
<xsl:function name="func:isEmbellished" as="xs:boolean">
<xsl:param name="node"/>
<xsl:param name="mode"/>
  ...
</xsl:function>
```

Description

This function implements the rules, from the MathML specification, that determine if an element is an embellished operator.

See

`http://www.w3.org/TR/2003/REC-MathML2-20031021/chapter3.html#id.3.2.5.7`

Parameters

node Element to check.

mode Stretching mode: `v` is to retrieve the operators that stretch vertically (default value), `h` is for horizontally, and `b` is for both vertically and horizontally.

Returns

Returns `true` if an element is an embellished operator.

Name

math:maction (in formatting mode) — Represents an action that reacts at a user sollicitation.

Synopsis

```
<xsl:template match="math:maction" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
  ...
</xsl:template>
```

Description

This element is not fully supported yet. To implement a default behaviour, the first child of the `maction` element is computed by using its formatting mode. Therefore, the `maction` node will not be annotated and will be replaced by its first child node.

Name

math:mfenced (in formatting mode) — Represents an expression enclosed by fences and separated by operators.

Synopsis

```
<xsl:template match="math:mfenced" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
  ...
</xsl:template>
```

See

`mfencedCompose`

Description

`mfenced` is an element that can be replaced by a `mrow` composed of two or more `mo` elements and its children. For example

Example 5. mfenced: original code

```
<mfenced open="[" close="]" separators=";|">
  <mn>1</mn>
  <mn>2</mn>
  <mn>3</mn>
</mfenced>
```

can be replaced by:

Example 6. mfenced: replacement code

```
<mrow>
  <mo fence="true">[</mo>
  <mn>1</mn>
  <mo separator="true">;</mo>
  <mn>2</mn>
  <mo separator="true">|</mo>
  <mn>3</mn>
  <mo fence="true">]</mo>
</mrow>
```

This example will render like that, with:

Example 7. mfenced: renderer

[1; 2|3]

We can see in this example the three optionnal arguments of `mfenced`:

`open` Determines the opening fence of the expression. The default value is **(**.

`close` Determines the closing fence of the expression. The default value is **)**.

`separators` Determines a sequence of one character separator that will be used to separate each children of `mfenced`. The default value is **,**. If there are not enoough separators to separate each child, the last one is repeated.

The formatting mode will transform the `mfenced` element into an `mrow` as mentionned above, and finally call the formatting mode of the `mrow` on it. First, all attributes are retrieved and spaces in `separators` atribute are deleted. A `mrow` node is created containing the opening and the closing `mo` and a composition of children and separators. This composition is done calling the `mfencedCompose` template. This template takes two arguments: the child nodes and the `separators` string attribute without space.

Finally, the formatting mode of the newly created `mrow` is called to compute and annotate it.

Name

mfencedCompose

Synopsis

```
<xsl:template name="mfencedCompose">
<xsl:param name="elements"/>
<xsl:param name="separators"/>
  ...
</xsl:template>
```

Description

This recursive template adds the first element and, if it is not the last element, the first separator is added too into an `mo` element with the `separator` attribute set to `true`. Then, the template is called again with elements and separators left. If it lefts only one separator, the recursion will always be called with that separator.

Parameters

elements	Child elements that compose the `mfenced`.
separators	Separators to add between two consecutive elements.

Returns

Returns the new composed row of elements.

Name

isPrime — Checks if an element is a prime token.

Synopsis

```
<xsl:function name="func:isPrime" as="xs:boolean">
<xsl:param name="node"/>
  ...
</xsl:function>
```

See

`http://www.nabble.com/RE%3A-Rendering-primes%3A-<msup><mi>x<-mi><mo>--x2032-<-mo><-msup>-p18157100.html`

Description

This function check if the node element is a prime. Such an element, as a superscript, has not to be shifted and script level must remain the same as the base. The characters that return true with this function are asterisk (x2a), degree (xb0), prime (x2032), double prime (x2033), back prime (x2035) and double back prime (x2036).

Parameters

node Element to check.

Returns

Returns `true` if an element is an prime operator.

Name

math:msup (in formatting mode) — Formatting a superscript.

Synopsis

```
<xsl:template match="math:msup" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
<xsl:param name="rightSwitch" tunnel="yes"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

Description

`msup` element has two children: the first child is the base and the second is the superscript. After font size computation, the base and the superscript are computed by calling the formatting mode template on the first and second child. The superscript gets a `X` coordinate value that depends on the base's size in order to place its box on the right of base one. Some general parameters are also modified when the superscript computation is called: the display style has to be `false` and the script level has to be incremented by one. Using this new value, the size of superscript elements will be smaller than base ones.

After that, some information is retrieved for each child: its height, `Y` coordinate of its top edge and its height over its baseline. This data will be used to compute the final height, baseline and coordinate of the box.

The next four variables are used to compute a shift value for the superscript. By default, this value depends on the base height, if the base element is lower than 1.2em, the shift value will be 80 percent of the base height over the baseline. In all other case, the default value will be 90 percent of the base height over the baseline. If the users as specified a shift value, using the `superscriptshift` attributes, this value will be retrieved and used instead of the default one. Then the shift value is corrected with respect to the initial position of the superscript and finally the descender of the superscript is added to the final shift value if it is not a token element.

After that, the box representation is computed by using the shift value. The height is computed by taking the difference between the lowest and the highest `Y`. The baseline is the baseline of the base.

Fnally, the tree node is annotated and contains, like all other elements, its box representation, and information that determine if the `msup` is an embellished operator (`EMBELLISH`, `LSPACE`, `RSPACE`, `ACCENT` and `stretchVertical`. Some other information is also added to shift and to place the superscript: `SHIFTY_BASE` that will shift the base on the y-axis to place it correctly if necessary, `SHIFTY_SUPERSCRIPT` that will shift the superscript on the y-axis to its final position, and `SHIFTX_SUPERSCRIPT` that will withdraw the `LSPACE` value of the superscript if this one is an embellished operator. This last shift on x-axis is done to draw the superscript much closer to its base.

Name

math:msub (in formatting mode) — Formatting a subscript.

Synopsis

```
<xsl:template match="math:msub" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="rightSwitch" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

Description

`msub` element has two children: the first child is the base and the second is the subscript. It works quite the same way as `msup`. The first difference is the computation of the initial `X` coordinate for the subscript child. This value is computed using the right bearing value of the base if this last one has such a value. This computation is done in order to place the subscript elements closer to the base. Other differences appear in the computation of subscript shift value. This value depends on the subscript height and not on the base height like the superscript in the `msup` element. Initially, the shift is 50 percent of the subscript height over the baseline. If the user specified the `subscriptshift` attribute, it will be retrieved and used instead of the initial value. A last correction is added which depends on the initial positionnement of the subscript.

The tree node is annotated with the same information as `msup`. However, only the subscript element gets shift attributes: `SHIFTY_SUBSCRIPT` and `SHIFTX_SUBSCRIPT` that have the same role that in the `msup` element.

Name

math:msubsup (in formatting mode) — Formatting both a superscript and a subscript.

Synopsis

```
<xsl:template match="math:msubsup" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
<xsl:param name="rightSwitch" tunnel="yes"/>
<xsl:param name="overUnderSpace" tunnel="yes"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

Description

`msubsup` element has three children: the first child is the base, the second is the subscript and the third is the superscript. The formatting mode is a composition of the `msup` and `msub` formatting mode. The superscript has the same computation as in `msup` element and subscript as in `msub`. The shift values for the scripts are also the same. A difference appears before computing the box representation, an other shift value is computed if the superscript covers the subscript. In this case, both superscript and subscript have to be shifted to remove this covering.

The box representation is then computed and finally, the tree is annotated exactly the same way as for both `msup` and `msub` elements.

Name

math:mover (in formatting mode) — Formatting an overscript.

Synopsis

```
<xsl:template match="math:mover" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
<xsl:param name="overUnderSpace" tunnel="yes"/>
  ...
</xsl:template>
```

Description

It consists of two children: the base is the first child and the overscript is the second. First, the base is computed and information about its box is retrieved to achieve further computation. Due to `accent` behaviour, adjustement has to be done to compute the overscript. First, the `accent` attribute is retrieved. If no value has been entered, accent is set to `false`. Then, the overscript is computed and, after that, the accent value is recomputed by using the `ACCENT` attributes of the overscript if this one is an embellished operator. If the new `accent` value differs from the old one, the overscript have to be recomputed.

These two passes have to be done because the `accent` attribute modifies the computation of the overscript element. If `accent` is `true`, the overscript has to be closer and the `scriptlevel` is not modified. On the other hand, when `accent` is `false`, the `scriptlevel` for the overscript element has to be incremented by one. Therefore, this element will have a smaller font size.

After overscript computation, some information about its box is retrieved to compute the `mover` final box.

A shift value is then computed to move the overscript away from the base. This value is zero if the `accent` attribute is `true`, if not, this space shift is taken from the `overUnderSpace` global parameter.

The box representation of the `mover`box is then computed. The height is the sum of the base height, the overscript height and the shift value previously computed. The baseline is the base's one and, since the overscript has to be drawn above the base, then the upper left corner `Y` coordinate is the `Y` coordinate of the overscript top edge.

Finally, the tree node is annotated with the box representation, with the embellished operator attributes and with shift values:

`SHIFTX_BASE` This shift value is used to centre the base horizontally with the overscript if the base is smaller than the overscript.

`SHIFTX_OVERSCRIPT` This shift value is used to centre the overscript horizontally with the base if the overscript is smaller than the base.

`SHIFTY_BASE` This value is used to place the base at its final place. The overscript has to be placed above the base. Therefore, the base has to be shifted down on the y-axis.

`SHIFTY_OVERSCRIPT` The overscript has to be shifted up on the y-axis to be drawn over the base.

Name

math:munder (in formatting mode) — Formatting an underscript.

Synopsis

```
<xsl:template match="math:munder" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
<xsl:param name="overUnderSpace" tunnel="yes"/>
  ...
</xsl:template>
```

Description

It consists of two children: the first is the base and the second is the underscript. `munder` is computed exactly the same way as the `mover` element. Only some variable names change, typically, `overscript` is replaced by `underscript`. The shift values are also computed differently because the `underscript` has to be drawn under the base and not over it.

Name

math:munderover (in formatting mode) — Formatting both an underscript and an overscript.

Synopsis

```
<xsl:template match="math:munderover" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
<xsl:param name="overUnderSpace" tunnel="yes"/>
  ...
</xsl:template>
```

Description

It consists of three children: the base is the first, the underscript is the second and the overscript is the third. This element is formatted as a combination of both an `mover` and an `munder` element. It first computes the base, then the underscript and finally the overscript. These two last elements are computed in two passes to handle correctly the `accent` attributes. These two passes are done the same way as for `mover` element.

The box representation is then computed. The height is the sum of each element's height plus the overscript and the underscript shift value. The width is the width of the largest element among the base, the overscript and the underscript. The baseline is the base's one and the upper left corner `Y` is the `Y` coordinate of the overscript box top edge.

The tree is finally annotated with box representation and with all shift values from both `mover` and `munder` elements. The x-axis shift values are computed to center each element.

Name

math:mfrac (in formatting mode) — Formatting a fraction.

Synopsis

```
<xsl:template match="math:mfrac" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="displayStyle" tunnel="yes"/>
<xsl:param name="fracWidMarg" tunnel="yes"/>
<xsl:param name="numDenSpace" tunnel="yes"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

Description

It consists of two children: the first is the numerator and the second is the denominator. First, like all other elements, the numerator and the denominator are computed using the corresponding formatting mode template. When calling the template, some parameters have to be modified to follow the specification. If the display style is `false`, the script level has to be incremented by one, and, if it is `true`, it has to be set to `false`.

The width, height and bottom edge `Y` coordinate of each child is then retrieved to help compute the box representation of the fraction.

After that, `mfrac` attributes is retrieved:

`linethickness` Determines the size of the fraction bar. By default, this value is 1. After retrieving it, the line thickness is computed in pixel using the `unitInPx` function. A value with no unit determines a multiplication of the thin value, for example the default value is 1, without unit, it means that the fraction bar must have a height of `1 * thin`. It is why the thin space literal is computed in pixels before computing the final fraction bar height. This value will be used as default value for the `unitInPx` function.

`numalign` Determines the alignement of the numerator. Values can be `center`, `left` or `right`. The default one is `center`.

`denomalign` Determines the alignement of the denominator. Values can be `center`, `left` or `right`. The default one is `center`.

A shift value is also computed to place the fraction bar, this value is computed from the baseline. The fraction bar has to be aligned with a minus sign, in the middle of the text. Therefore, the half size of letter **x** is used.

The box representation is then computed. The `width` is the maximum between the numerator and the denominator width plus a margin value both on the right and on the left from the global parameters (`fracWidMarg`). The baseline and the bottom of the box is set.

Finally, the tree is annotated with the box representation and with shift value for the numerator and the denominator:

`SHIFTXNUM` Represents an x-axis shifting to place the numerator with respect to the `numalign` attribute.

`SHIFTXDEN` Represents an x-axis shifting to place the denominator with respect to the `denomalign` attribute.

`SHIFTYNUM` Represents a y-axis shifting to place the numerator above the fraction bar to its final position.

`SHIFTYDEN` Represents a y-axis shifting to place the denominator under the fraction bar to its final position.

Values to place and draw the fraction bar are also added to the annotated tree: `FRAC_BAR_Y` is the `Y` coordinate of the fraction bar and `FRAC_BAR_HEIGHT` is its size.

Name

math:msqrt (in formatting mode) — Formatting a square root.

Synopsis

```
<xsl:template match="math:msqrt" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="rtFrnSpcFac" tunnel="yes"/>
<xsl:param name="rtTopSpc" tunnel="yes"/>
  ...
</xsl:template>
```

Description

Its children constitute a row and must be treated using the same mechanisms that for the `mrow`. First of all, the children are computed using the `subMrow` template (like an `mrow`). A space is added before the children to allow drawing of the square root symbol in front of them. The space value is computed using `rtFrnSpcFac` value from the global parameters with respect to the current font size.

The box representation is then computed the same way as in an `mrow` element. However, in opposition to `mrow` a space is added on the top of the box to draw the square root line over the child elements. This value is coming from the global parameters (`rtTopFac`).

Finally, the tree is annotated the same way as an `mrow` element.

Name

math:mroot (in formatting mode) — Formatting a n-ary root.

Synopsis

```
<xsl:template match="math:mroot" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="scriptlevel" tunnel="yes"/>
<xsl:param name="rtFrnSpcFac" tunnel="yes"/>
<xsl:param name="rtTopSpc" tunnel="yes"/>
  ...
</xsl:template>
```

Description

It consists of two children: the first one is the base and the second is the index. First, the base child and the index child are computed and some information about their boxes is retrieved (size and position) for further computation. The base initial `X` coordinate is shifted to the right to add space for drawing the root symbol. After that, the box representation is computed using child information.

Finally, the tree is annotated with the box, with information about the size and the place of the radical (`RADICAL_HEIGHT` and `RADICAL_Y`) and with shift values for the children:

`SHIFTY_INDEX` Determines a y-axis shifting to place the index.

`SHIFTY_BASE` Determines a y-axis shifting to place the base.

Name

math:mtable (in formatting mode) — Formatting a table.

Synopsis

```
<xsl:template match="math:mtable" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="tableSpace" tunnel="yes"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

See

`computeStretch`, `stretchRows`, `mtableWidth`, `mtableShiftY` and `mtableShiftX`.

Description

This element has one or more `mtr` elements as children. It is also quite complex to render, elements that compose a column have to be aligned. The elements on a line also have to be aligned. The table must be centered on the middle of the mathematic expression. The main idea is to first compute all the cells on the same place, as if they were not in a table. For example, in the two by two identity matrix, all 0 and 1 are computed like simple `mn` element. Finally, shift values are computed to move each cell to its final position.

Therefore, in the formatting mode, the first action is to compute all `mtr` children by calling the appropriate template in formatting mode. After that, cells that contain a stretchy operator have to be stretched with respect to other cells that compose the columns and the row. For example, if a column contains a cell with a right arrow, given that the arrow has to stretch horizontally, the width of this cell has to have the value of the largest cell in the column. To compute the new size of the cells, stretch values are computed using the `computeStretch` function and the cell nodes are modified by using the `stretchRows` template.

After that, the box representation is computed. The height is the sum of each row's height plus spaces between each two lines. The space size value comes from the global paramaters (parameter `tableSpace`). The table width is computed by calling the `mtableWidth` template on rows. The baseline is placed at the middle of the table. And, the upper left corner `Y` is the `Y` coordinate of the table top edge.

The `columnalign` is then retrieved. This attributes is used to determine how the cells in a column have to be aligned. The default value is `center`. The shift values for the y-axis are computed by using the `mtableShiftY` template and the shift values for the x-axis are computed by using the `mtableShiftX` template and `columnalign` attribute.

Finally, the tree is annotated with the box representation and all shift values.

Name

computeStretch

Synopsis

```
<xsl:template name="computeStretch">
<xsl:param name="rows"/>
<xsl:param name="i" select="1"/>
<xsl:param name="j" select="1"/>
  ...
</xsl:template>
```

See

`isEmbellished`

Description

All cells are handled by using `i` and `j` parameters as if the template were two loops. However, given that XSLT does not provide loop command, the two loops are done by using a recursion scheme.

For each cell, the width is the largest cell in the column if the element in the current cell has to be stretched horizontally, zero in all other cases. The height is the highest cell in the row if the element in the current cell has to be stretched vertically, zero in all other cases. To check if an element has to be stretched, the `isEmbellished` template is used.

Parameters

rows	Row children from a table.
i	Column index of the current element. By default, this index is 1.
j	Row index of the current element. By default, this index is 1.

Returns

As output, the template provides a list of height and width for each cell. Rows are delimited by a semi-colon in that list. For example, in the following table $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, the output will be `1.width 1.height 2.width 2.height ; 3.width 3.height 4.width 4.height`. The value for width is zero if the content of the cell has not to be stretched horizontally, if the content has not to be stretched vertically, the height is zero.

Name

mtableWidth

Synopsis

```
<xsl:template name="mtableWidth">
<xsl:param name="rows"/>
<xsl:param name="i" select="1"/>
<xsl:param name="width" select="0"/>
<xsl:param name="tableSpace" tunnel="yes"/>
  ...
</xsl:template>
```

Description

For each column, the width of the largest cell in the column and space size between two cells (`tableSpace` global parameter) are added to `width`. The recursion is called with an `i` incremented by one and the newly computed width. The final output (when there is no more column to treat) is the value of the accumulator `width` minus one space size between to cell.

Parameters

rows — Row children from a table.

i — Index of the current column that is handled. By default, this index is 1.

width — Accumulator that contains the width for a table composed of the `(i-1)`th first columns of all the row children.

Returns

Returns the total width of a table.

Name

mtableShiftY

Synopsis

```
<xsl:template name="mtableShiftY">
<xsl:param name="y"/>
<xsl:param name="rows"/>
<xsl:param name="tableSpace" tunnel="yes"/>
  ...
</xsl:template>
```

Description

Recursion is done over the rows set. At each step, a shift value is computed for the first row in the set by using the difference between its final `Y` position (given in parameter) and its current `Y` position. The recursion is called for the rest of the set with an updated `Y` value. This new value is computed by using the height of the first row and size of space between two cells (`tableSpace` global parameter).

Parameters

rows Row children from a table.

y Final top edge `Y` coordinate of the first row in the `rows` parameters.

Returns

As output, it provides a sequence of values that represent the shift for all rows: `(1st row shift, 2nd row shift, ..., last row shift)`.

Name

mtableShiftX

Synopsis

```
<xsl:template name="mtableShiftX">
<xsl:param name="rows"/>
<xsl:param name="columnalign"/>
<xsl:param name="i" select="1"/>
<xsl:param name="j" select="1"/>
<xsl:param name="width" select="0"/>
<xsl:param name="tableSpace" tunnel="yes"/>
  ...
</xsl:template>
```

Description

The recursion is done the same way as in the `computeStretch` template.

For each cell, the alignement value (`center`, `left` or `right`) is retrieved from, ordered by preference, `mtd` element (retrieved by using XPath on the rows parameter), `mtr` element (also retrieved by using XPath on the rows parameter) or `mtable` element (given in parameter). If no value is specified by the user in `mtd`, `mtr` nor `mtable`, the default alignement value, coming from the `mtable` element, is `center`.

After computing the alignement value, the largest element in the current column and the width of the current cell are retrieved. These values are then used to compute a shift value with respect to the alignement value. The width accumulator is used to determine the initial shift value to place the current cell in its final column.

Parameters

rows	Row children from a table.
columnalign	`columnalign` `mtable` attribute formatted as a sequence.
i	Column index of the current element. By default, this index is 1.
j	Row index of the current element. By default, this index is 1.
width	Accumulator that contains the width for a table composed of the `(i-1)`th first columns of all the row children.

Returns

As output, it provides a sequence of shift value for each cell, the row are separated by a semicolon. It is done the same way as in the `computeStretch` template. For example, in the following table $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, the output will be `1.shiftValue 2.shiftValue ; 3.shiftValue 4.shiftValue`.

Name

math:mtr (in formatting mode) — This element represent a row of a table.

Synopsis

```
<xsl:template match="math:mtr" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="tableSpace" tunnel="yes"/>
  ...
</xsl:template>
```

See

`alignRow`

Description

`mtr` element is computed like a basic `mrow` element. It is composed by one or more `mtd` children.

After computing the current font size, the cells that compose the row are computed by using the `alignRow` template in order to align all the cells on the same baseline. After that, the box representation of the row is computed. The height is the difference between the highest and the lowest `Y` coordinate among all the children. The width is the sum of all the cells width plus a space between them (using the `tableSpace` global parameter). The baseline is the lowest baseline among children and upper left corner `Y` coordinate is the lowest `Y` coordinate among all the children.

The `columnalign` is then retrieved. The default value is `inherited` if no one is specified.

Finally, the tree is annotated by using the box representation, the shift value —as it is computed in an `mrow` element—, and with `COLUMNALIGN` attributes.

Name

alignRow

Synopsis

```
<xsl:template name="alignRow">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
<xsl:param name="nodes"/>
<xsl:param name="firstNode" select="0"/>
  ...
</xsl:template>
```

See

`alignChild`

Description

This template has the same behaviour as the `alignChild` template from `mrow`. It calls the formatting mode on the current element and aligns it on the first child baseline. This child is found with the `firstNode` parameter.

Parameters

x, y, baseline	Formatting mode required parameters.
nodes	Elements handled in the template.
firstNode	Determines if the current element is the first one in the row. It will give its baseline to all other elements.

Returns

Returns all cells aligned on the same baseline.

Name

math:mtd (in formatting mode) — This element represents a cell in a row. It has the same behaviour as a row element.

Synopsis

```
<xsl:template match="math:mtd" mode="formatting">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="baseline" select="0"/>
  ...
</xsl:template>
```

Description

The formating mode is exaclty the same as the `mrow` element. The only difference is the attribute `columnalign` that has to be retrieved. The default value is `inherited` if none is specified.

Name

stretchRows

Synopsis

```
<xsl:template name="stretchRows">
<xsl:param name="rows"/>
<xsl:param name="stretchValues"/>
  ...
</xsl:template>
```

See

`computeStretch`

Description

This template simply calls `mtr` stretch mode template on each row. This new template mode is used to modify the width and the height of a node. It works directly in the annotated tree and changes the value of `WIDTH` and `HEIGHT` annotation.

Parameters

rows	Rows of a table to correct.
stretchValues	Values computed by the `computeStretch` template.

Returns

Returns the corrected rows.

Name

math:mtr (in stretch mode) — Correct width and height on a row.

Synopsis

```
<xsl:template match="math:mtr" mode="stretch">
<xsl:param name="stretchValues"/>
  ...
</xsl:template>
```

See

`computeStretch`, `stretchCols`

Description

First, all its `mtd` elements are recomputed by using the `stretchCols` template. After that, the annotated tree is recomposed by using these new cells.

Parameters

stretchValues	Sequence that represents the new width and height (computed by the `computeStretch` template) for all the cells in that row.

Name

stretchCols

Synopsis

```
<xsl:template name="stretchCols">
<xsl:param name="rows"/>
<xsl:param name="stretchValues"/>
  ...
</xsl:template>
```

Description

It simply recomputes all `mtd` elements by calling their stretch mode template with new width and height as paramaters.

Parameters

rows	All cells that will be corrected.
stretchValues	New width and height values (in a sequence) for all these cells.

Name

math:mtd (in stretch mode) — Correct width and height on a cell.

Synopsis

```
<xsl:template match="math:mtd" mode="stretch">
<xsl:param name="width"/>
<xsl:param name="height"/>
  ...
</xsl:template>
```

See

`computeStretch`

Description

This template simply copies the annotated node and changes `WIDTH` and `HEIGHT` annotations if the parameters are not equal to zero.

Parameters

width	New width for the element.
height	New height for the element.

Drawing mode

Name

math:mi|math:mn|math:mtext|math:ms (in draw mode) — Drawing a token.

Synopsis

```
<xsl:template match="math:mi|math:mn|math:mtext|math:ms" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

Description

To draw these elements in the SVG file, the SVG `text` element is used. The placement of this element is done using the coordinates of the baseline's reference dot. To compute these coordinates, the height over baseline value is added to the upper left corner`Y` coordinate.

Name

math:mspace (in draw mode) — Drawing a space.

Synopsis

```
<xsl:template match="math:mspace" mode="draw"/>
```

Description

Nothing has to be drawn with these element. Therefore, an empty template has been created in the XSLT stylesheet.

Name

math:mo (in draw mode) — Drawing an operator.

Synopsis

```
<xsl:template match="math:mo" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

See

`drawHorizontalDelimiter` and `drawVerticalDelimiter`

Description

First, the final coordinate is computed according to shift and left space values. There are three different display ways of an operator with respect to the direction of stretch. If the operator has to stretch vertically, the `drawVerticalDelimiter` is called, if the operator has to stretch horizontally, the `drawHorizontalDelimiter` is called. In all other cases, the operator is drawed like other token elements (`mi`, `mn`, etc.).

The operator that has to be composed is grouped into a SVG `g` element with a common style attribute. This group will simplify the drawing of the stretched operator parts in `drawXxxDelimiter` template.

Name

math:math|math:mrow|math:merror|math:mphantom|math:menclose|math:mstyle (in draw mode) — Drawing a box.

Synopsis

```
<xsl:template match="math:math|math:mrow|math:merror|math:mphantom|math:menclose|m
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

See

`drawEnclose`

Description

All the children of a row are grouped in a SVG `g` tag that represents a group of elements on the canvas. The style attribute is set on this tag to determine the default style of the box. After writing this tag, the drawing mode template of each child is called in order to draw them, except if the element is a `mphantom`. The children of a `mphantom` element are never drawn. These children are shifted on the y-axis if necessary (attribute `SHIFT`).

If the element is an `merror` element, a box is drawn around child elements using the SVG `rect` tag that draws a rectangle.

If the element is an `menclose` element, the `drawEnclose` template is called to write decoration around child elements. This template takes five parameters: `X` and `Y` coordinates, `WIDTH` and `HEIGHT` of the row and `NOTATION` attribute that is transformed into a sequence to handle multiple notation.

Name

drawEnclose

Synopsis

```
<xsl:template name="drawEnclose">
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="width"/>
<xsl:param name="height"/>
<xsl:param name="notations"/>
<xsl:param name="errorMargin" tunnel="yes"/>
  ...
</xsl:template>
```

Description

This template will browse all notations and draw the appropriate `line` and `rect` SVG tag corresponding to the notation. The circle is drawn using the SVG `ellipse` tag to draw an ellipse and the `longdiv` notation is drawn using the SVG `path` element to draw a curve. This element is used to draw complex paths on the canvas.

Parameters

`x, y`	`X` and `Y` coordinates of the box that is decorated.
`width`	Width of the box that is decorated.
`height`	Height of the box that is decorated.
`notations`	Notation that will decorate the box

Name

math:msup (in draw mode) — Drawing a superscript.

Synopsis

```
<xsl:template match="math:msup" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

The drawing of this element is very simple. It calls the drawing mode template of each child by adding the shift values that have been computed in the formatting mode.

Name

math:msub (in draw mode) — Drawing a subscript.

Synopsis

```
<xsl:template match="math:msub" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

The drawing mode is exactly similar to the `msup` element. Only the name of the shift attributes differ.

Name

math:msubsup (in draw mode) — Drawing both a superscript and a subscript.

Synopsis

```
<xsl:template match="math:msubsup" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

The drawing mode simply draws the base, the subscript and the superscript are drawn by calling the draw mode template of each child. The shift values are added when calling these templates.

Name

math:mover (in draw mode) — Drawing an overscript.

Synopsis

```
<xsl:template match="math:mover" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

The drawing mode consists simply of drawing the children of `mover` by calling their template in drawing mode. The shift values are added in the call to correctly place each element.

Name

math:munder (in draw mode) — Drawing an underscript.

Synopsis

```
<xsl:template match="math:munder" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

Like the formatting mode, the drawing mode of `munder` element is exactly the same as the `mover` one, except some variable names.

Name

math:munderover (in draw mode) — Drawing both an overscript and an underscript.

Synopsis

```
<xsl:template match="math:munderover" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

All children are simply drawn by using the corresponding drawing mode template with the corresponding shift values.

Name

math:mfrac (in draw mode) — Drawing a fraction.

Synopsis

```
<xsl:template match="math:mfrac" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

The drawing mode is quite simple. It draws each child by calling the drawing mode on the children using computed shift values from the tree. Finally, it draws a line for the fraction bar using the SVG `line` element, `Y` coordinate and `height` from the annotated node.

Name

math:msqrt (in draw mode) — Drawing a square root.

Synopsis

```
<xsl:template match="math:msqrt" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
<xsl:param name="rtFrnSpcFac" tunnel="yes"/>
  ...
</xsl:template>
```

Description

The drawing mode first draws each child by calling the corresponding template in the drawing mode. After that, the square root symbol has to be drawn in front of the children and a line is added over them. It is done by using four SVG `line` elements.

Name

math:mroot (in draw mode) — Drawing a n-ary root.

Synopsis

```
<xsl:template match="math:mroot" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
<xsl:param name="rtFrnSpcFac" tunnel="yes"/>
  ...
</xsl:template>
```

Description

First, the base and the index are drawn using the appropriate template in the drawing mode. Shift values are also added to `Y` coordinate to draw them in the correct place. After that, the root symbol is drawn the same way as in the `msqrt` element.

Name

math:mtable (in draw mode) — Drawing a table.

Synopsis

```
<xsl:template match="math:mtable" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

See

`drawRows`

Description

To apply the correct shift values on each cell, the drawing mode calls an other template: `drawRows`. This template will draw each row by using the computed shift values.

Name

drawRows

Synopsis

```
<xsl:template name="drawRows">
<xsl:param name="rows"/>
<xsl:param name="shiftY"/>
<xsl:param name="shiftX"/>
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

This template calls `mtr` drawing mode template on each row by using the correct shift value. The y-axis shift value is direclty added to the `yShift` parameters. The x-axis values for a row are transformed into a sequence and given to the template through the `shiftX` parameter.

Parameters

rows	Sequence of `mtr` elements.
`shiftX`	Sequence of x-axis shift values for each cell in each row.
`shiftY`	Sequence of y-axis shift value to move rows to their final place.
`xShift and yShift`	Required parameters of drawing mode template.

Name

math:mtr (in draw mode) — Drawing a row in a table.

Synopsis

```
<xsl:template match="math:mtr" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
<xsl:param name="shiftX"/>
  ...
</xsl:template>
```

See

`drawCols`

Description

To apply the correct shift values on each cell, the drawing mode calls an other template: `drawCols`. This template will draw each cell by using the computed shift values.

Name

drawCols

Synopsis

```
<xsl:template name="drawCols">
<xsl:param name="rows"/>
<xsl:param name="shiftX"/>
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

This template calls `mtd` drawing mode template on each cell by using the correct shift value. The x-axis shift value is direclty added to the `xShift` parameters.

Parameters

`rows`	Sequence of `mtd` elements.
`shiftX`	Sequence of x-axis shift values for each cell in this row.
`xShift and yShift`	Required parameters of drawing mode template.

Name

math:mtd (in draw mode) — Drawing a cell of a table.

Synopsis

```
<xsl:template match="math:mtd" mode="draw">
<xsl:param name="xShift"/>
<xsl:param name="yShift"/>
  ...
</xsl:template>
```

Description

This mode behaves exactly like the `mrow` drawing mode. It calls the drawing mode of all its children.

Name

drawVerticalDelimiter

Synopsis

```
<xsl:template name="drawVerticalDelimiter">
<xsl:param name="delimiter"/>
<xsl:param name="height"/>
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="fontSize"/>
<xsl:param name="variant"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

See

`findBestSize` and `drawVerticalExtenser`

Description

First of all, a verification must be done to know if the operator has to be stretched. If it does not have to stretch, it will be simply drawn like a non-stretchy operator. After that, another verification checks if the operator can be composed, stretched or not. This verification is done using the two structures: `delimPart` and `delimScale`. If the operator cannot be composed or stretched, it will simply be centered by using a SVG `text` element. If the operator is in the `delimPart` structure, the operator will be composed, if the operator is in the `delimScale` structure, the operator will be scaled.

Compose

The number of parts needed to compose the symbols is retrieved from the structure. This number will be used to know which type of operator will be composed. After that, the index of part in the structure will be computed and the bounding box of each part will be retrieved from the metrics. A correction is done to avoid the small gaps on the canvas. The next line computes the number of extensers that will be added and the final font size of the operator calling the function `findBestSize`. A new font size is computed in order to have a round number of parts.

After all these computations, the bottom and the top parts of the operator will be drawn using a SVG `text` element. The top delimiter is only drawn if there are more than two parts or if the `extenser` attribute is `bottom`. In the same way, the bottom delimiter is only draw if there is more than two parts or if the `extenser` attribute is `top`. Now, the extenser has to be drawn and the way to draw them depends on the number of parts.

If the operator has four parts (like a curly bracket for example), a middle part is then added using a `text` element and two groups of extenser are drawn around this middle part using `drawVerticalExtenser` function. The extenser is only drawn if the number returned by `findBestSize` is bigger than zero.

If the operator has two or three parts, the extenser will be added if the number returned by `findBestSize` is bigger than zero. If the operator has two parts and if the extensers have to be drawn on the top, the `Y` coordinate has to be on the top of the box. In the other cases, it has to be under the top part of the operator.

Scale

First, a `scale` factor is computed and then the operator is drawn in a `text` box that is transformed using the SVG `transform` attribute and a `scale` transformation. The `Y` coordinate has to be corrected because the `scale` transformation modifies the coordinate system.

Parameters

`x, y`	`X` and `Y` coordinates of the bottom left corner of the operator box.
`delimiter`	Operator to stretch.
`height`	Total height of the box that the operator has to fill.
`fontSize`	Initial font size of the operator.
`variant`	Font variant for the operator.

Name

drawVerticalExtenser

Synopsis

```
<xsl:template name="drawVerticalExtenser">
<xsl:param name="n"/>
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="extenser"/>
<xsl:param name="extenserSize"/>
<xsl:param name="fontSize"/>
<xsl:param name="rotate" select="false()"/>
  ...
</xsl:template>
```

Description

It simply draws an extenser, then, if `n` is greater than one, the function is called again with next `Y` coordinate and decremented `n`.

Parameters

`n`	Number of extensers to draw
`x`	`X` coordinate for extensers
`y`	`Y` coordinate for the bottom extenser
`extenser`	Extenser to draw
`extenserSize`	Extenser size (in em)
`fontSize`	Size of the font to draw the extenser
`rotate`	`true` if the extenser has to be rotated

Name

drawHorizontalDelimiter

Synopsis

```
<xsl:template name="drawHorizontalDelimiter">
<xsl:param name="delimiter"/>
<xsl:param name="width"/>
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="fontSize"/>
<xsl:param name="variant"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

See

`findBestSize` and `drawHorizontalExtenser`

Description

The `drawHorizontalDelimiter` template is similar to the vertical one except in composition of operator. Some operators have to be composed with a vertical part that is rotated. In SVG, the rotation is done using the `transform` attribute and a `rotate` transformation. The modification that is done against the vertical composition is the check of the `hrotate` attributes of the `delimPart` that indicates if parts have to be rotated. And, using the `hrotate` values, the measure of the part that must retrieve the height if the parts are rotated, or the width of parts if not.

The other modifications concern adding the rotate transformation when the parts are drawn and calling the `drawHorizontalExtenser` template instead of the `drawVerticalExtenser` one.

Parameters

`x, y`	`X` and `Y` coordinates of the bottom left corner of the operator box.
`delimiter`	Operator to stretch.
`width`	Total width of the box that the operator has to fill.
`fontSize`	Initial font size of the operator.
`variant`	Font variant for the operator.

Name

drawHorizontalExtenser

Synopsis

```
<xsl:template name="drawHorizontalExtenser">
<xsl:param name="n"/>
<xsl:param name="x"/>
<xsl:param name="y"/>
<xsl:param name="extenser"/>
<xsl:param name="extenserSize"/>
<xsl:param name="fontSize"/>
<xsl:param name="rotate" select="false()"/>
<xsl:param name="fontName" tunnel="yes"/>
  ...
</xsl:template>
```

Description

This template is similar to `drawVerticalExtenser` except that the `Y` coordinate remains the same through the recursive call and `X` is incremented to the next coordinate at each template call.

Parameters

`n`	Number of extensers to draw
`x`	`X` coordinate for extensers
`y`	`Y` coordinate for the bottom extenser
`extenser`	Extenser to draw
`extenserSize`	Extenser size (in em)
`fontSize`	Size of the font to draw the extenser
`rotate`	`true` if the extenser has to be rotated

Name

findBestSize — Finds a round number of extensers that will cover a space with a font size as near as possible to the initial font size.

Synopsis

```
<xsl:function name="func:findBestSize" as="xs:double+">
<xsl:param name="height"/>
<xsl:param name="fontSize"/>
<xsl:param name="minPart"/>
<xsl:param name="partsSize"/>
<xsl:param name="extenserSize"/>
  ...
</xsl:function>
```

Description

This function follows the following algorithm:

1. If `$height <= $partsSize * $fontSize` then return `(0, $height div $partsSize)`
2. Else
   a. Compute: `$rawRatio = ($height - $partsSize * $fontSize) div ($fontSize * $extenserSize) + $minPart` and `$roundRatio = round($rawRatio)`
   b. Compute `$ratio`:
      i. If `$minPart < 3` or `$roundRatio` is even then `$ratio = $roundRatio`
      ii. Else
         A. If `$rawRatio < $roundRatio` alors `$ratio = $roundRatio - 1`
         B. Else `$ratio = $roundRatio + 1`
   c. Return `($ratio - $minPart, $height div (($ratio - $minPart) * $extenserSize + $partsSize))`

The Step 2.b is used to obtain an even number of extensers when the operator has a middle part. The number of extensers at the top (or on the left) of the middle part must be equal to the number of extensers at the bottom (or on the right) of this part.

Parameters

`height`	Total height to cover
`fontSize`	Initial font size
`partsSize`	Required part size (in em) (all element excepts the extenser)
`extenserSize`	Extenser size (in em)
`minPart`	Required number of part

Returns

Returns a sequence of two numbers: the first is the number of extensers to add and the second is the new computed font size.

Font metrics stylesheet

Introduction

This stylesheet offers functions to interact with an XML FOP file metrics. Currently, three types of FOP metrics are supported: WinAnsiEncoding, Type1 font and TTF font. The last one is prefered because the metrics contain more symbol metrics and more precise metrics. WinAnsiEncoding and Type1 files only contain a maximum of 255 metrics.

Name

findFont — Finds an existing font metrics file for a font name with respect to variants (italic, bold, etc.)

Synopsis

```
<xsl:function name="func:findFont">
<xsl:param name="font"/>
<xsl:param name="variant"/>
  ...
</xsl:function>
```

Description

Firstly, if a metrics file exists for the current font name and variant, the name of this file is returned. If not, a check is proceeded to simplify the variant.

If the variant is `-Bold-Italic`, a metrics file is searched for the `-Bold` and the `-Italic` variant. If one of them exists, the name of this metrics file is returned. Otherwise, a metrics file with no variant is checked and returned if it exists.

If the variant is only `-Bold` or only `-Italic`, a check for a file with no variant is proceeded. If it succeeds, the name of this metrics file is returned.

In all other cases, when no font metrics file can be found, an empty name is returned.

Parameters

font	Font name
variant	Variant for the font, this variant can be `-Italic`, `-Bold`, `-Bold-Italic` or empty.

Returns

Returns the name of the metrics file (without extension) or an empty string if no font was found.

Name

findWidth — Find the width of a character from a list of fonts. The first font of the list that contains the character will be used.

Synopsis

```
<xsl:template name="findWidth">
<xsl:param name="name"/>
<xsl:param name="fonts"/>
<xsl:param name="variant"/>
<xsl:param name="fontInit" select="$fonts"/>
  ...
</xsl:template>
```

Description

Firstly, a check is done to verify if the character is not an invisible operator such as invisible time or apply function. If it is one, the size 0 is returned. Otherwise, the character is checked among the font list.

The list is browsed to find a metrics file (using `findFont` function) that contains the character. If such a file can be found, the width metric from this file is returned. Otherwise size 0.8 is returned.

To retrieve a width from a metric file, the template `findWidthFile` is used.

Parameters

name	Character to check.
fonts	Font list that is used to find the character width.
variant	Variant for the font, this variant can be `-Italic`, `-Bold`, `-Bold-Italic` or empty.

<varlistentry>fontInit
Initial font list that is used to find the character width. Since the font list will be modified through the recursion. The initial list has to be saved.
</varlistentry>

Returns

Returns the width of the character in em or 0.8em if the character is not found within the font list.

Name

findWidthFile — Find the width of a character from a font metrics file.

Synopsis

```
<xsl:template name="findWidthFile">
<xsl:param name="name"/>
<xsl:param name="fontName" select="'STIXGeneral'"/>
  ...
</xsl:template>
```

Description

Firstly, the final font name is computed by adding the extension `.xml` to the `fontName` parameter, and the character code point is retrieved by using the XPath `string-to-codepoints` function. The metrics file document tree is then retrieved by using the `document` function.

After that, the width attribute is retreived from the metrics document with respect to the font metrics encoding. If the encoding is `CID` encoding, a glyph start index (`gs`) and unicode start value (`us`) are computed to retrieve the attribute `w` (which contains the width of the character) from the `(gi + 1 + codePoint - us)`th `wx` element of the metrics file. In all other cases (WinAnsiEncoding), the `wdt` attribute (which contains the width of the character) from the `char` element whose its `idx` attribute is `codePoint`.

Finally, if this width is zero, the width of **x** is returned instead. If no width was found, -1 is returned and, in all other cases, the width divided by 1000 is returned.

Parameters

name	Character to find.
fontName	Name of the font metric file (without extension).

Returns

Returns the width of the character in em or -1 if the character is not found in the font metrics file.

Name

findBbox — Find the bounding box of a character from a list of fonts. The first font of the list containing the character will be used.

Synopsis

```
<xsl:function name="func:findBbox" as="xs:double+">
<xsl:param name="name"/>
<xsl:param name="fonts"/>
<xsl:param name="variant"/>
  ...
</xsl:function>
```

Description

This function simply calls the `findBbox` template. It is used because functions are easier to call in some cases than a template.

See

`findBbox`

Parameters

name	Character to check.
fonts	Font list that is used to find the character width.
variant	Variant for the font, this variant can be `-Italic`, `-Bold`, `-Bold-Italic` or empty.

Returns

Returns the bounding box of the character in em or `(0, 0, 0, 0)` if the character is not found within the font list. The bounding box is returned in a sequence of four elements: `(xMin, xMax, yMin, yMax)`.

Name

findBbox — Finds the bounding box of a character from a list of font. The first font of the list that contains the character will be used.

Synopsis

```
<xsl:template name="findBbox" as="xs:double+">
<xsl:param name="name"/>
<xsl:param name="fonts"/>
<xsl:param name="variant"/>
<xsl:param name="fontInit" select="$fonts"/>
  ...
</xsl:template>
```

Description

Firstly, a check is done to verify if the character is not an invisible operator such as invisible time or apply function. If it is one, the bounding box `(0, 0, 0, 0)` is returned. Otherwise, the character is checked among the font list.

The list is browsed to find a metrics file (using `findFont` function) that contains the character. If such a file can be found, the bounding box metrics from this file is returned. Otherwise sequence `(0, 0, 0, 0)` is returned.

To retrieve a bounding box from a metric file, the template `findBboxFile` is used.

Parameters

name	Character to check.
fonts	Font list that is used to find the character bounding box.
variant	Variant for the font, this variant can be `-Italic`, `-Bold`, `-Bold-Italic` or empty.
fontInit	Initial font list that is used to find the character bounding box. Since the font list will be modified through the recursion. The initial list has to be saved.

Returns

Returns the bounding box of the character in em or `(0, 0, 0, 0)` if the character is not found within the font list. The bounding box is returned in a sequence of four elements: `(xMin, xMax, yMin, yMax)`.

Name

findBboxFile — Finds the bounding box of a character from a font metrics file.

Synopsis

```
<xsl:template name="findBboxFile" as="xs:double+">
<xsl:param name="name"/>
<xsl:param name="fontName" select="'STIXGeneral'"/>
  ...
</xsl:template>
```

Description

Firstly, the final font name is computed by adding the extension `.xml` to the `fontName` parameter, and the character code point is retrieved by using the XPath `string-to-codepoints` function. The metrics file document tree is then retrieved by using the `document` function.

After that, the bounding box attribute is retrieved from the metrics document with respect to the font metrics encoding. The bounding box can only be retrieved in the CID encoding. Therefore, the glyph start index (`gs`) and unicode start value (`us`) are computed to retrieve the attributes `xMin`, `xMax`, `yMin` and `yMax`, from the `(gi + 1 + codePoint - us)`th `wx` element of the metrics file. With a metrics file encoded in WinAnsiEncoding, bounding box `(0, 0, 0, 0)` is returned.

If the character cannot be found in the metrics file, the value -1 is returned.

Parameters

name	Character to find.
fontName	Name of the font metric file (without extension).

Returns

Returns the bounding box of the character in em or -1 if the character is not found in the font metrics file. The bounding box is returned in a sequence like that `(xMin, xMax, yMin, yMax)`

Name

findHeight — Finds height and depth of a string from a list of font by using the bounding box (from metrics) of each character in the string. The first font of the list containing the character will be used.

Synopsis

```
<xsl:function name="func:findHeight" as="xs:double+">
<xsl:param name="str"/>
<xsl:param name="fonts"/>
<xsl:param name="variant"/>
  ...
</xsl:function>
```

Description

This function simply calls `findHeightAlt` template.

See

`findHeightAlt`

Parameters

str	String to check.
fonts	Font list that is used to find the character width.
variant	Variant for the font, this variant can be `-Italic`, `-Bold`, `-Bold-Italic` or empty.

Returns

Returns the height and depth of a string in a sequence: `(height, depth)`.

Name

findHeightAlt — Finds height and depth of a string from a list of font by using the bounding box (from metrics) of each character in the string. The first font of the list containing the character will be used.

Synopsis

```
<xsl:template name="findHeightAlt">
<xsl:param name="str"/>
<xsl:param name="strLen"/>
<xsl:param name="i" select="1"/>
<xsl:param name="fonts"/>
<xsl:param name="variant"/>
<xsl:param name="height" select="0"/>
<xsl:param name="depth" select="0"/>
  ...
</xsl:template>
```

See

`findBbox`

Description

For each character, the bounding box is retrieved and the template is called recursively with an updated value for `height` and `width`. For the height value, the maximum between `yMax` (from the bounding box) and the `height` paramater is taken, and for the depth, the minimum between `yMin` (from the bounding box) and the `depth` parameter is taken.

Parameters

str	String to check.
strLen	Number of characters in the string.
i	Index of the character that is currently analysed.
fonts	Font list that is used to find the character width.
variant	Variant for the font, this variant can be `-Italic`, `-Bold`, `-Bold-Italic` or empty.
height	Accumulator that saves the highest height for the first `i` characters.
depth	Accumulator that saves the lowest depth for the first `i` characters.

Returns

Returns the height and depth of a string in a sequence: `(height, depth)`.