The MMT Language and System

Florian Rabe
Jacobs University Bremen

August 29, 2011

1 Syntax

The MMT language and its semantics have been described in depth in [Rab09]. Here we will focus on describing the XML syntax to someone who already has a general understanding of MMT.

The semantics of notations is described in Sect. 2.

1.1 URIs

We define three data types for addressing MMT elements: the type *MMTURI* of MMT URIs and the types *Name* and *QName* of unqualified and qualified MMT names. They are defined by the following grammar:

Unqualified name	*Name*	::=	$pchar^+$
Qualified name	*QName*	::=	$Name(/Name)^*$
MMT URI	*MMTURI*	::=	$D \mid M \mid S$
Document URI	D	::=	*URI*, no query, no fragment
Module URI	M	::=	$D?QName \mid ?(/Name)^*$
Symbol URI	S	::=	$M?QName \mid ??(/Name)^*$
	URI, *pchar*		see RFC 3986 [BLFM05]

We will use Q and R to range over qualified names. In a URI $D[?Q[?R]]$, D, Q, and R are called the document, module, and symbol part. A URI is called **absolute** if the document part is an absolute URI and if — if present — neither Q nor R are of the form $(/Name)^*$. Otherwise, it is called **relative** or an MMT URI **reference**.

Note that a *pchar* may be any character permitted in a URI except for "/", "?", "#" "[", "]", and "%". Furtermore, all percent-encoded characters are permitted.

Example 1. In this example, we abbreviate `http://cds.omdoc.org/algebra/algebra.omdoc` with `A`.

`O/algebra/algebra.omdoc?monoid?unit` is an absolute symbol URI. It refers to the symbol `unit` declared in the module `monoid` declared in the document `A`.

The /-character in the module part separates submodules. `A?monoid/latex` refers to the module `latex` declared within the module `A?monoid` (e.g., a module containing notations to render monoids in Latex syntax).

The /-character in the symbol part separates named imports, called structures in MMT. Let `A?group?mon` refer to an import `mon` declared within the module `A?group` that imports the module `A?monoid`. Then `A?group?mon/unit` refers to the symbol `A?monoid?unit` imported via this import. In general, if the symbol part of an MMT URI has n components, then the first $n-1$ must be the names of named imports.

The grammar is unambiguous. Every absolute MMT URI has exactly one of the following three forms: $D?Q?R$, $D?Q$, or D. Every relative MMT URI has exactly one of the following six forms:

relative to ...	1. ... document URI	2. ... module URI	3. ... symbol URI
Document reference	d		
Module reference	$d?q$	$?q_{ref}$	
Symbol reference	$d?q?r$	$?q_{ref}?r$	$??r_{ref}$

Here we distinguish three kinds of relative URIs according to whether they are relative to a document, a module, or a symbol URI. For these three cases, the resolution of MMT URI references u against an absolute base URI U is defined as follows:

1. u starts with a URI reference d: If D' is the result of resolving d against the document part of D of U according to RFC 3986, then the resulting MMT URI is D', $D'?q$, or $D'?q?r$ depending on the form of u.
   The special case where d is empty implies $D' = D$. (Beware that software packages for the URI data type such as in Java 1.5 might implement the obsolete RFC 2396, where empty d was resolved as $D' = D/...$)

2. u is of the form $?q_{ref}$ or $?q_{ref}?r$ where q_{ref} is of the form $(/Name)^*$: If U is of the form $D?Q$ or $D?Q?R$, then u is interpreted relative to $D?Q$ yielding $D?Qq_{ref}$ or $D?Qq_{ref}?r$, respectively. It is an error if U is a document URI. In particular, the special cases where q_{ref} is empty yield $D?Q$ and $D?Q?r$, respectively.

3. u is of the form $??r_{ref}$ where r_{ref} is of the form $(/Name)^*$: If $U = D?Q?R$ is a symbol URI, then the resolution yields $D?Q?R/r$. It is an error if U is a module or document URI. In particular, the special case where r_{ref} is empty yields $D?Q?R$.

Example 2. Assume a base URI `http://cds.omdoc.org/algebra/algebra.omdoc?group?mon`. The following table gives examples of resolutions of relative URIs for each of the above six cases. Here we abbreviate `http://cds.omdoc.org` with `O`.

URI	Resolution
`mathml.omdoc`	`O/algebra/mathml.omdoc`
`?group`	`O/algebra/algebra.omdoc?group`
`../logics/fol/fol.omdoc?fol?and`	`O/logics/fol/fol.omdoc?fol?and`
`?/latex`	`O/algebra/algebra.omdoc?group/latex`
`?/latex?circ`	`O/algebra/algebra.omdoc?group/latex?circ`
`??/unit`	`O/algebra/algebra.omdoc?group?mon/unit`

In addition to the above grammars, we introduce the following convention: MMT URIs that contain less than two occurrences of ?, can also be written with 2 $?s$ by appending ?. In other words, $D??$ and $D?Q?$ abbreviate D and $D?Q$, respectively.

Thus, (recalling that no ?-character may occur in URIs that have no query component) every MMT URI can be written uniquely as a ?-separated triple. The components of this triple are a URI and two lists of strings, and empty lists may be used to represent the absence of a component.

1.2 Document Level Elements

Label: `omdoc` (a document unit)

Attributes:

name	type	value
`name`	*MMTURI*	the optional name of the unit
`base`	*MMTURI*	the base URI for the unit's content, relative to base URI given by parent, empty by default

Children: `omdoc* & xref* & module*`

Comment: If this occurs as the root of a document that has a URL, then the name must be omitted or be equal to the last segment of that URL's path.

1.3 Module Level Elements

Label: `theory` (a theory)

Attributes:

name	type	value
`base`	*MMTURI*	the base URI of the module, relative to the base URI given by the parent, empty by default
`name`	*Name*	the name of the view
`meta`	*MMTURI*	the URI of the optional meta-theory, relative to base URI given by parent

Children: `(include* & symbol*) | definition{theory}`

Comment: Here, a symbol can be a constant, a structure, or a notation. Definitions are actually not supported yet and only added for symmetry.

Label: `view` (a theory morphism, postulated link)

Attributes:

name	type	value
`base`	*MMTURI*	the base URI of the module, relative to the document base, empty by default
`name`	*Name*	the name of the view
`from`	*MMTURI*	the domain of the view, relative to base URI given by parent
`to`	*MMTURI*	the codomain of the view, relative to base URI given by parent

Children: `(include* & assignment*) | definition{morph}`

Comment: An assignment can be an assignment to a constant or an assignment to a structure.

Label: `style` (a named set of notations)

Attributes:

name	type	value
`base`	*MMTURI*	the base URI of the module, relative to the document base, empty by default
`name`	*Name*	the name of the style
`for`	*MMTURI*	the base URI used for `for` attributes, relative to the module's base URI, empty by default
`defaults`	`use` | `ignore`	defaults to `use`, determines the treatment of default notations given in theories

Children: `(include* & notation*) | definition{notset}`

Comment: Definitions are actually not supported yet and only added for symmetry.

All module level elements are named. The base URI of a module defaults to the document URI. However, a differing URI may be provided with the `base` attribute of the module or an ancestor. The latter should only be used in generated documents because it prevents reference by location.

A module with name n and base URI B is addressable via the module URI $B?n$. Module URIs (but not module names) must be unique within a file.

A document unit with name n whose parent is addressable as D is addressable as D/n. Document unit names must be unique within a document unit.

1.4 Symbol Level Elements

Some symbol level elements are named (notations optionally so). If a symbol with name n occurs in a module with URI M, the symbol is addressable via the URI $M?n$.

1.4.1 Elements in Theories

Label: `include` (inclusion of a theory into the containing theory)

Attributes:

name	type	value
`from`	*MMTURI*	the domain of the inclusion, relative to containing theory

Children:

Label: `constant` (e.g., a sort, function, predicate, judgment, or proof rule)

Attributes:

name	type	value
`name`	*Name*	the name of the constant

Children: `type{term}?  & definition{term}?`

Label: `structure` (named instantiation of another theory, definitional link)

Attributes:

name	type	value
`name`	*Name*	the name of the structure
`from`	*MMTURI*	the domain of the structure, relative to containing theory

Children: `(include* & assignment*) | definition`{`morph`}

Comment: An assignment can be an assignment to a constant or an assignment to a structure.

Label: `alias` (alternative name for another symbol)

Attributes:

name	type	value
`name`	*Name*	the name of the alias
`for`	*MMTURI*	the symbol the alias references

Children:

Comment: Contrary to a defined symbol, an alias is transparent and does not introduce a new symbol.

1.4.2 Elements in Links

Label: `include` (inclusion of a morphism into the containing link)

Children: `morph`

Label: `conass` (assignment to a constant in a link)

Attributes:

name	type	value
`name`	*QName*	the name of the constant

Children: `term`

Comment: The child is the term that is assigned to the constant.

Label: `strass` (assignment to a structure in a link)

Attributes:

name	type	value
`name`	*QName*	the name of the structure

Children: `morph`

Comment: The child is the morphism that is assigned to the constant.

1.4.3 Elements in Styles

Label: `include` (inclusion of a style into the containing style)

Attributes:

name	type	value
`from`	*MMTURI*	the domain of the inclusion, relative to base URI of containing style

Children:

Every notation has a role. The permitted values of `role` are given in Sect. 2. There are two ways to give notations, direct and via parameters:

Label: `notation` (a notation)

Attributes:

name	type	value
`name`	*Name*	the optional name of the notation
`for`	*MMTURI*	the optional URI to which the notation applies, relative to base URI of style
`role`	string	the simple role to which the notation applies
`wrap`	`true | false`	a flag specifying whether the notation is merged with more specific ones
`precedence`	$\mathbb{Z}^*$	the optional output precedence

Children: `pres`

Comment: A notation without a name has no URI. `precedence` may only be given if the role is bracketable.

Label: `notation` (a notation)

Attributes:

name	type	value
`name`	*Name*	the optional name of the notation
`for`	*MMTURI*	the optional URI to which the notation applies, relative to base URI of style
`role`	string	the simple role to which the notation applies
`precedence`	$\mathbb{Z}^*$	the optional output precedence
`fixity`	string	the optional fixity: `pre | post | in | inter | bind`
`application-style`	string	the optional application style: `math | lc`
`associativity`	string	the optional associativity: `none | left | right`
`implicit`	$\mathbb{N}$	the number of implicit arguments

Children:

Comment: A notation without a name has no URI. `precedence` may only be given if the role is bracketable.

1.5 Fragment References

In addition, document fragments defined elsewhere may be referenced. Documents with references are semantically identical to those where references are replaced with their resolution except that the appropriate `base` attribute is added after resolving.

Label: `X` (a reference to a X element)

Attributes:

name	type	value
`href`	*MMTURI*	the URI of the referenced element

Children:

Comment: `X` may be any document, module, or symbol level label.

Comment: These elements may carry additional attributes if their key and value are identical to those in the referenced element.

1.6 Object Level Elements

1.6.1 Terms

The type `term` represents MMT terms. These are given as OpenMath objects wrapped in an OMOBJ element. Furthermore, morphism application of μ to ω is encoded in OpenMath using the special theory MMT with base MMT:

<OMA>
  <OMS base="MMT" module="mmt" name="morphism-application"/>
  μ
  ω
</OMA>

1.6.2 Morphisms

The type `morph` represents MMT morphisms. Morphisms are links, identity, and composition, which are encoded in OpenMath using the special theory MMT. We use MMT to abbreviate http://omdoc.org/mmt.

Link $D?Q$:

<OMS base="D" module="Q"/>

Composition $\mu_1 \ldots \mu_n$:

<OMA>
  <OMS base="MMT" module="mmt" name="composition"/>
  μ_1
  ...
  μ_n
</OMA>

Identity id_T:

<OMA>
  <OMS base="MMT" module="mmt" name="identity"/>
  T
</OMA>

Theories are encoded like links.

1.6.3 Presentations

The type `pres` represents presentations. These are lists of *presentation elements* that are used to define structural translations of MMT expressions into other formats or languages. A presentation is evaluated relative to a list of MMT expressions, and this evaluation returns a string or an XML element. Syntax and semantics of presentations are described in Sect. 2.

1.7 Resolving MMT URIs

Document level The scheme and authority of a URI D must resolve to some root directory. Then the path of D is resolved relative to that directory. For the resolution of paths, we treat the paths ending in / and those not ending in / as identical.

A path P is resolved relative to the directory D as follows:

- P is empty, then resolve to D.
- If $P = p/P'$ and p is a subdirectory of D, then resolve P' relative to D/p.
- If $P = p/P'$ and p is a file in D, then resolve P' relative to the content of that file (which must be an `omdoc` element).

A path P is resolved relative to an `omdoc` element O as follows:

- P is empty, then resolve to O.
- If $P = p/P'$ and p is the value of a `name` attribute of an `omdoc` child of O, then resolve P' relative to that child.

Note that this makes the file structure transparent in the following sense. Assume a directory D with files $n_1, \ldots, n_r$ containing the respective `omdoc` element O_i. Let O be the file containing

```
<omdoc>
  <omdoc name="n_1">
    O_1
  </omdoc>
  ...
  <omdoc name="n_r">
    O_r
  </omdoc>
</omdoc>
```

Then the resolutions of a path relative to D and O are the same.

We can implement this transparency with a standard apache web server: In every directory D add a file `index.omdoc` containing O as defined above and add a directive in the `.htaccess` file to serve `index.omdoc` as the directory listing. (If D should happen to contain a file `index.omdoc` already, we can pick any other file name for it.) Then apache's path resolution will correspond to the above definition for all paths that do not end in /.

The connection to directory listings is stressed if we use this alternative – semantically equivalent – definition of O:

```
<omdoc>
  <xref target="n_1"/>
  ...
  <xref target="n_r"/>
</omdoc>
```

Note that it is always legal to split an `omdoc` file into directories. However, the opposite is only legal if module names are unique across the directory.

Module Level A module level URI $D?Q$ is resolved by resolving D to an `omdoc` element O and then resolving Q relative to it as follows:

- If $Q = q$ is a single segment, then resolve to the module with name q in O. Here, the modules in O are the module level children of `omdoc`-descendants of O.
- If $Q = q/Q'$, then resolve Q' relative to the `omdoc`-wrapped resolution q.

Here we assume that there are no `base` attributes set in O.

Symbol Level A module level URI $D?Q?R$ is resolved by resolving $D?Q$ to a module level element M and then resolving R relative to it as follows:

- If $R = r$ is a single segment, then resolve to the symbol with name r in M. Here, the symbols in M are the symbol level children of `omdoc`-descendants of M.
- If $R = r/R'$, then resolve R' relative to the domain of the structure with name r in M and translate the result along said structure.

Note that the structure of nested `omdoc` elements is transparent to the resolution of modules and symbols. Thus, the uniqueness of module and symbol names, which is necessary to make resolution well-defined, nested `omdoc` elements must be disregarded.

2 Notations

2.1 Presentable Expressions

A **presentable** MMT expression is any expression produced from any non-terminal symbol of the MMT grammar. Most presentable expressions are characterized by

- a role, which refers to the non-terminal symbol from which it is produced,
- a components, which is a list of presentable expressions that occur in the first production.
- a path, an *MMTURI* identifying or describing the expression (if possible) or its main component,

For example, the expression $@(f, a_1, \ldots, a_n)$ for the application of f to arguments $a_1, \ldots, a_n$ is produced using the production $\omega ::= @(\omega, \ldots, \omega)$. Its role is `application`, and its components are $f, a_1, \ldots, a_n$. If f is a constant, then the path is the *MMTURI* of that constant.

Roles and Components In the following we list all presentable expressions with their roles and components. We will use :: and *nil* for head-tail composition of lists and + to append an element to the end of a list. For optional parts, a component $[C]$ means that the components is either C or has the special value *None*.

Expression	Role	Components
$T \stackrel{[M]}{=} \{\vartheta\}$	`Theory`	$T :: [M] :: \vartheta$
$l : S \to T \stackrel{[\mu]}{=} \{\sigma\}$	`View`	$l :: S :: T :: [\mu] :: \sigma$
$s : S \to T = \mu$	`DefinedView`	$l :: S :: T :: \mu :: nil$
$c : [\tau] = [\delta]$	`Constant`	$c :: [\tau] :: [\delta] :: nil$
$s : S \stackrel{[\mu]}{=} \{\sigma\}$	`Structure`	$s :: S :: [\mu] :: \sigma$
$s : S = \mu$	`DefinedStructure`	$s :: S :: \mu :: nil$
$c \mapsto \omega$	`ConAss`	$c :: \omega :: nil$
$s \mapsto \mu$	`StrAss`	$s :: \omega :: nil$
$x : \tau = \delta$	`Variable`	$x :: \tau :: \delta :: nil$
$g?T?c$	`constant`	$g :: T :: c :: nil$
x	`variable`	see below
$g?T?s$	`structure`	$g :: T :: s :: nil$
$g?l$	`view`	$g :: l :: nil$
$g?T$	`theory`	$g :: T : nil$
$\top$	`hidden`	nil
ω^μ	`morphism-application` (b)	$\omega :: \mu :: nil$
$@(\omega_1, \ldots, \omega_n)$	`application` (b)	$\omega_1 :: \ldots :: \omega_n :: nil$
$\beta(\omega_1; \Upsilon; \omega_2)$	`binding` (b)	$\omega_1 :: \Upsilon + \omega_2$
$\alpha(\omega_1; \omega_2 = \omega_3)$	`attribution` (b)	$\omega_2 :: \omega_1 :: \omega_3 :: nil$
$\mu_1 \bullet \ldots \bullet \mu_n$	`composition` (b)	$\mu_1 :: \ldots :: \mu_n :: nil$
id_T	`identity` (b)	$T :: nil$
toplevel structural expression	`Toplevel`	see below
toplevel object expression	`toplevel`	see below

Bracketable roles are those for which rendering will produce brackets based on input and output precedences. They are marked with (b). Only notations for those roles may have a `precedence` attribute, which defaults to 0 if omitted.

There are two special cases:

- A variable occurrence has three components: its name, its de-Bruijn index, and the id of the content expression (`OMATTR` or `OMV`) where the variable is bound (see Sect. 2.2.3 for IDs).
- The role `Toplevel` is chosen for every structural expression immediately after the translation algorithm is called from the outside (as opposed to recursive calls occurring during the translation). Its only component is the expression itself. Using this role, it is possible to include header and footer into the result of the translation.
- The role `toplevel` is similar to the above, but used whenever the toplevel of an object is translated (from the outside or by recursion). This permits to wrap all objects in some way if the output format requires it (e.g., a `math` element when generating presentation MathML). This role has two components: the object itself and its OpenMath XML representation. In the latter component, all subexpressions have unique XML IDs (see Sect. 2.2.3 for IDs).

Paths The meaning of the path depends on the presentable expression, or more strictly its role. The path is defined as follows:

- For all structural roles, the path is the *MMTURI* of the expression, e.g., $g?T$ for a theory T declared in a document g.

- For all roles that refer to an expression, it is the *MMTURI* of that expression. This applies to the roles `constant`, `structure`, `view`, `theory`.
- For roles of composed objects, paths are computed recursively as follows:
  - `variable`: none
  - `morphism-application`: the path of the applied morphism,
  - `application`: the path of the applied function,
  - `binding`: the path of the binder,
  - `attribution`: the path of the key,
  - `identity`: the path of the theory,
  - `hidden`: none,
  - `composition`: The path of the first morphism,
  - values and foreign objects: none
- For the roles `Toplevel` and `toplevel`, the path is the path of the toplevel expression.

2.2 Syntax and Semantics of Presentations

In the following we will define the well-formed presentation elements and their semantics. We write $[m, n]$ for the set of all integers between and including m and n and $\mathbb{Z}^*$ for the set $\mathbb{Z} \cup \{\texttt{infinity}, \texttt{-infinity}\}$.

For a presentation element P, its evaluation $R(P, C, i)$ is parametric in a list $C = C_0, \ldots, C_{n-1}$ of MMT expressions and a value $i \in \{0, \ldots, n-1\}$. The evaluation returns a string or a list of XML elements.

For a list of presentation elements $P_1 \ldots P_n$, the evaluation $R(P_1 \ldots P_n, C, i)$ is the concatenation $R(P_1, C, i) + \ldots + R(P_n, C, i)$. If any of these returns an XML element, the concatenation is the concatenation of XML elements where all string components are treated as XML text nodes; consecutive text nodes are merged. Otherwise, it is the concatenation of strings. (This concatenation is associative.)

2.2.1 Producing Literal Values

Label: `text` (produce a string)

Attributes:

name	type	value
`value`	string	the string to produce, defaults to the empty string

Children:

Evaluation: This presentation element is evaluated as the value of the `value` attribute.

Label: `element` (produce an XML element)

Attributes:

name	type	value
`prefix`	string	the namespace prefix of the element, default to the empty string
`name`	string	the label of the element, defaults to the empty string

Children: `pres & attribute*`

Evaluation: This presentation element is evaluated as the XML element with namespace prefix and label as given by the `prefix` and `name` attributes. If the former is empty, the element has no namespace prefix. The list of children of the produced XML element is the evaluation of the children of the presentation element except for the `attribute` elements. The attributes of the XML element are given by the evaluation of all the `attribute` children.

Label: `attribute` (produce an XML attribute)

Attributes:

name	type	value
`prefix`	string	the namespace prefix of the attribute, default to the empty string
`name`	string	the label of the attribute, defaults to the empty string

Children: `pres`

Comment: No child may be an `element`.

Evaluation: An `attribute` element is evaluated as the XML attribute with namespace prefix and name as given by the `prefix` and `name` attributes. If the former is empty, the attribute has no namespace prefix. The value of the attribute is the evaluation of its children.

2.2.2 Recursing into Components

Label: `components` (iterate through C)

Attributes:

name	type	value
`begin`	$\mathbb{Z}$	the begin index b, defaults to 0
`end`	$\mathbb{Z}$	the end index e, defaults to -1
`step`	$\mathbb{Z} \setminus \{0\}$	the step size s, defaults to 1

Children: `separator{pres}? & main{pres}?`

Comment: If `separator` is not present, it defaults to an empty element. If `main` is not present, it defaults to `<main><recurse/></main>`.

Evaluation: Let S and M be the list of children of the `separator` and `main` elements. Intuitively, this presentation elements evaluates M for all components from b to e with step size s, and puts the evaluation of S in between.

Formally, putting $\overline{S} := R(S, C, i)$, the evaluation is defined as

$$R(M, C, b') + \overline{S} + R(M, C, b' + s) + \overline{S} + \ldots + \overline{S} + R(M, C, b' + ls)$$

where $b' \in [0, n-1]$, $e' \in [0', n-1]$, and (i) if $s > 0$, then $b' \leq e'$ and l is the largest natural number such that $b' + ls \leq e'$, and (ii) if $s < 0$, then $b' \geq e'$ and l is the smallest natural number such that $b' + ls \geq e'$.

b' and e' are obtained by the following computation: If $n = 0$, or if $b \notin [-n, n-1]$, or $e \notin [-n, n-1]$, an error is issued. Otherwise, b and e are taken modulo n to obtain b' and e'. Now if $e - b$ has a different sign from s, an error is issued. Then the above conditions hold.

Label: `recurse` (recursively translate a component)

Attributes:

name	type	value
`precedence`	$\mathbb{Z}^*$	the relative input precedence for the recursion, defaults to 0
`offset`	$\mathbb{Z}$	the offset relative to the current component, defaults to 0

Children:

Comment: A precedence may not be given if within a notation for a structural role.

Evaluation: This presentation element evaluates to $T(C_{i+o}, [p])$ where p is the value of the `precedence` attribute and o is the value of the `offset` attribute.

Example 3. For example, let P be the presentation element

```
<components begin="0" end="-1" step="2">
  <separator><newline/></separator>
  <main><text value="Component number "/><index/><text value=": "/><recurse/>
</components>
```

Then we have $R(P, A :: B :: C :: D :: E :: F :: nil, i)$ yields

Component number 0: A'
Component number 2: C'
Component number 4: E'

where A', C', and E' denote the recursive renderings of A, C, and E.

Label: `component` (recurse into a single component)

Attributes:

name	type	value
`index`	$\mathbb{Z}$	the index of the component
`precedence`	$\mathbb{Z}$	the relative input precedence of the recursion, defaults to 0

Children:

Comment: A precedence may not be given if within a notation for a structural role.

Evaluation: This is a shortcut for a `components` elements where `begin` and `end` index are j and the only child is `recurse precedence="`p`"` where j and p are the values of the `index` and `precedence` attribute. This means that the evaluation is $T(C_j, [p])$.

Label: `index` (render the position of a component)

Attributes:

name	type	value
`offset`	$\mathbb{Z}$	the offset relative to the current component, defaults to 0

Children:

Evaluation: This presentation element evaluates to the result of $i + o$ as a string where o is the value of the offset attribute.

2.2.3 Miscellaneous Elements

Label: `id` (produces a unique ID)

Children:

Evaluation: This evaluates to a string that uniquely identifies the currently translated expression. These IDs can be used to create arbitrary unique names required by the target format of the translation.

This ID is equal to the corresponding XML ID in the OpenMath expression occurring as the second component of the role `toplevel`. Thus, it can be used for parallel markup links from presentation to content when translation to presentation MathML.

Label: `ifpresent` (case distinction for omitted components)

Attributes:

name	type	value
`index`	$\mathbb{Z}$	the index of the tested component

Children: `then{pres}? else{pres}?`

Comment: If `then` or `else` are not present, they default to empty elements.

Evaluation: Let j be the value of the `index` attribute. If $C_i \neq \_$, this presentation element evaluates to the evaluation of the children of its `then` child, otherwise to the evaluation of the children of its `else` child.

Label: `hole` (a placeholder)

Attributes:

name	type	value
`index`	$\mathbb{Z}$	the index of the supplied argument within a list of arguments to fill the placeholder

Children: `pres`

Evaluation: This presentation element serves as a placeholder that is replaced during the dynamic computation of presentations. Even it is not filled, it evaluates to the evaluation of its children.

Label: `fragment` (a call to a template)

Attributes:

name	type	value
`name`	string	the template name

Children: `arg pres * | pres`

Comment: Children P of the form `pres` abbreviate a single `arg` child with children P.

Evaluation: This presentation element calls another notation, which is used as a template. It is evaluated by obtaining the presentation p for the notation key $(-, \texttt{fragment}\ :n)$ (see below)

where n is the value of the `name` attribute. If $p_0, \ldots, p_n$ are the presentations in the children, then this presentation element is evaluated to the evaluation of $p(p_0, \ldots, p_n)$.

The latter notation describes the filling of placeholders. For a presentation p and a list of presentations $p_0, \ldots, p_n$, we write $p(p_1, \ldots, p_n)$ for the presentation arising from p by replacing every placeholder with index $0 \leq\leq n$ in p with p_i. Placeholders with index $i > n$ or $i < 0$ are replaced with their respective children. For example, brackets are given as presentations in which a placeholder indicates the position of the bracketed expression.

2.3 Semantics of Notations

A notation is a tuple of a notation key and a presentation and possibly and output precedence. The notation key determines when the notation is used, the presentation determines the rendering produced form the notation.

A **notation key** is a tuple (u, r) of an optional *MMTURI* u and a role r. We write $u = -$ if it is omitted. If r is a bracketable role, then the notation must also give an **output precedence** that is used for bracket generation. The **presentation** is an expression in a simple language for text or XML output, it may contain references to components of the currently rendered object. For *declarative notations*, the presentation is computed from parameters such as fixity. The presentation has an optional output precedence.

The high-level structure of the presentation algorithm is as follows:

1. Input: a presentable expression E with optional *MMTURI* U, role R, and component list C, a style N, and an optional input precedence *iPrec*.
2. A notation n applicable to E is selected from N.
3. The presentation p of n is obtained.
4. If R is bracketable, depending R, *iPrec*, and the output precedence of n, P is wrapped in brackets yielding P'.
5. Output: the evaluation $R(P', C, 0)$ of P' in context C.

All steps are described below.

Selecting Notations Styles may have multiple or no notations for the same notation key. The following rules are used:

- A style may not locally declare two notations for the same key.
- A style has all locally declared notations, and all notations that imported style have, with one exception: Local notations shadow imported declarations for the same key.
- If U is omitted, n is the notation with role R in N. It is an error if no such notation exists.
- If U is given, then
  - if N has a notation for either (U, R) or for $(-, R)$, then that n is that notation,
  - if N has notations for neither (U, R) or $(-, R)$, it is an error.
  - if N has notations n_1 with presentation p_1 for (U, R) and n_2 with presentation p_2 for $(-, R)$ and n_2 has the `wrap` flag, then n is a notation with presentation $p_2(p_1)$ (see below for applying presentations); the precedence of n is that of n_1.
  - if N has notations n_1 for (U, R) and n_2 for $(-, R)$ but n_2 does not have the `wrap` flag, then $n = n_1$.

Obtaining Presentations If n gives a presentation directly, P is that presentation.

Otherwise, P is computed from the declarative parameters of n as follows:

1. The operator is the first component in C.
2. According to the number i of implicit arguments determined by n, the following i components are the implicit arguments, the remaining components the explicit arguments.
3. For pre- and postfix notations, the implicit and the explicit arguments are listed with the separator given by the fragment `argsep`. Then operator, implicit, and explicit arguments are used to fill the corresponding placeholders in the fragment `pre` or `post`, respectively.
4. For infix notations, the operator and the implicit arguments are used to fill the placeholders in the fragment `operimp`. Then the result is placed among the explicit arguments using the fragments `opsep` and `argsep`.

Bracketing If R is not bracketable, no brackets are generated and $P' = P$.

Otherwise, the $P' = b(P, d)$ where d is a below and b is the presentation of one of the following three fragments: `fragment:brackets`, `fragment:ebrackets` (elidable brackets), or `fragment:nobrackets`. While these fragment names a fixed meaning, it is still the style's responsibility to provide notations for them.

The selection among the three fragments is as follows:

- If *iPrec* is not given, no brackets. Thus, expressions can be forced to be unbrackets by not giving an input precedence. This is typical when recursing from structural levels into object levels.
- If *iPrec* is given, then the brackets depend on the difference $d = oPrec - iPrec$:
  - no brackets if $d = \infty$.
  - elidable brackets if $0 < d < \infty$.
  - brackets if $d \leq 0$.

Special Fragment Names The following table lists all fragment names that have a special meaning.

Fragment	Placeholders	Function
`brackets`	bracketed expressions	unelidable brackets
`ebrackets`	bracketed expressions, elision level	elidable brackets
`nobrackets`	unbracketed expression	no brackets
`argsep`	none	separator between arguments
`opsep`	none	separator between operator and arguments
`argsep`	none	separator between arguments
`pre`, `post`	operator, implicit arguments, explicit arguments	pre- and postfix notations
`operimp`	operator, implicit arguments	operator in infix notations

3 Archives

An MMT archive [HIJ+11] organizes connected documents in a file system structure. This is inspired by and similar to the project view in software engineering, specifically in Java. MMT provides archive-level functions for building and indexing document collections.

Dimensions MMT supports the following dimensions, which occur as toplevel folders in an archive:

- source: source files in any language
- compiled: one MMT file for every source file, following the same directory structure
- content: one file for every MMT module with their directory structure induced by their MMT URI
- narration: one MMT file for every source file, similar to the ones in compiled but containing links into the content instead of MMT modules
- relational: relational index
- mws: MathWebSearch [KŞ06] index

Metadata Archives may contain meta in a file `META-INT/MANIFEST.MF`. The syntax of this file is line-wise `key:value` pairs. Keys may occur only once. Predefined keys are:

- `id`: a string. This identifies the archive.
- `source`: a string. This identifies the source language and is used to choose a compiler.
- `narration-base`: a URI. URIs for the source/narration documents are formed by concatenating this URI with the path within the archive (excluding the dimension).
- `source-base`: a URI. A URI that is used by compilers as the base URI for source files.

Compilers Compiler can be provided as plugins to MMT. They are classes that implement the MMT compiler interface (see Sect. 4). This class defines an abstract file-to-file compile method. It also has a method to test applicability: This method is passed the `source` string above when finding a matching compiler for an archive.

Build Process MMT provides to build the dimensions of an archive, starting from the source:

- `compiled` is generated from `source` using a matching compiler.
- `content` and `narration` are generated generically from `content`.
- Indices are generated generically from `content` and `narration`.
- a `.mar` that packages all dimensions is generated generically.

4 The Shell

Invocation The MMT shell is invoked by

```
java -jar jomdoc.jar
```

Further command line parameters are passed to the shell and executed as a command. In particular, the command `file F` can be used to execute a startup file. After invocation, the shell can be controlled via STDIN/STDOUT.

Besides caching all loaded documents, the shell only maintains one state variable: a base MMT-URI. All paths are interpreted relative to this base.

General Syntax *Commands* are given by a keyword followed by a whitespace-separated list of arguments and terminated by a newline. Empty lines and lines starting with `//` are ignored.

We use the following meta-variables for command arguments:

- `F`: a file name that is interpreted relative to the current directory,
- `U`: an MMT-URI that is interpreted relative to the current base URI,
- `A`: an action that is executed on a path or object.

An exception are *actions*, where the keyword is placed after the first argument in the style of OO-programming, see below.

Basic Commands

- `log+ C`, `log- C` : Switch on/off logging of component `C`.
- `base U`: Sets the base path to `U`.
- `file F`: Reads the file `F` and executes every line as a command. If a command causes an error, execution is aborted.
- `exit`: Quits the shell.

Interacting with Documents The shell stores a set of documents that are parsed into abstract data structures and made available for querying.

- `read U`: Retrieves, parses, validates, and stores the document with URI `U`. If read documents contain notation definitions, these are parsed and stored as well. However, the handling of notations is lazy: Dereferencing of references to notation containers and parsing of found notations are on demand.
- `clear`: Deletes all read knowledge items from memory.
- `printAll`, `printXML`: Dumps the memory, used for testing.
- Documents are accessed using actions on MMT-URIs as described below.

If the execution of these commands, requires documents that have not been read yet, these are retrieved automatically. This happens, for example, if the read document imports a theory from another document, or if the requested path points to a document that has not been read yet.

Both reading and accessing documents is done via document URIs, not via URLs. Therefore, when reading documents, a catalog is necessary to translate URIs into URLs (see below).

Catalog Commands The catalog maintains a translation map from URIs to URLs. It is initially empty so that no addressing is possible. The following commands add entries to the catalog.

- `catalog F`: This is used to add local working copies to the catalog. It takes a file `F` in the locutor (see `https://locutor.kwarc.info/`) registry format and creates an entry for every working copy listed in it. The repository URLs are treated as URIs that are translated to the location of the local working copy. (See the example file `locutor.xml` file in the distribution.)
- `local`: This adds an entry for the local file system. URIs of the form `file:///U` are translated to themselves.
- `ombase F`: This creates a catalog entry for an OMBase server described in `F`. See the example file `ombase.xml` in the documentation.

Archive Commands The following commands permit the registration and manipulation of archives:

- `compiler C ARGS`: This registers a compiler. `C` is the URI of a Java class, which must extend `info.kwarc.mmt.api.backend.Compiler` and have a constructor that takes no arguments. `ARGS` is a whitespace-separated list of string. The compiler will be initialized by calling its `init` method, to which `ARGS` is passed as a Scala `List[String]`. Compilers may maintain their own data structures and auxiliary threads; if they do so, they must clean up after themselves in their `destroy` method to avoid memory leaks.
- `archive add F`: This registers an archive with local root folder `F`.
- `archive ID D F`: This builds the dimensions `D` in the archive with id `ID`, optionally restricted to the folder/file `F`. Legal values for `D` are `compile`, `content` (which will produce `content` and `narration`), `mws`, and `relational`.
- `archive mar F`: This builds the `mar` file and stores it in `F`.

Actions Actions provide a simple infix syntax to pipe retrieved knowledge items through some typed post-processing operations.

The command $O\ A\ a_1\ \ldots\ a_n$ evaluates O and then applies action A with additional arguments a_i. This corresponds to $O.A(a_1,\ldots,a_n)$ in OO-programming. Actions may be chained: $O\ A\ a_1\ \ldots\ a_n\ B\ b_1\ \ldots\ b_n$ corresponds to $O.A(a_1,\ldots,a_n).B(b_1,\ldots,b_n)$. Actions are classified according to the type of O, which is MMT-URI, MMT-Object, or Non-MMT-Object, and the return type, which is MMT-Object, Non-MMT-object, or Nothing.

Actions on MMT-URIs U

- empty action: dereferences U and returns it (MMT-Object)
- `closure`: dereferences U and returns the closure as a self-contained document (MMT-Object)
- `deps xml`: dereferences U and returns its dependency set in XML representation (Non-MMT-Object)
- `deps locutor`: dereferences U and returns its dependency set in locutor representation (Non-MMT-Object)

Actions on MMT-Objects O

- empty action: returns the text representation of O (Non-MMT-Object)
- `component C`: returns the component of O called `C` (MMT-Object)
- `xml`: returns the XML representation of O (Non-MMT-Object)
- `present U`: returns the rendering of O using style `U` (Non-MMT-Object)

Valid component names `C` are in particular `type` and `definition` if O is a constant.

Actions on Non-MMT-Objects, all returning Nothing:

- empty action: prints to standard output
- `write F`: prints to file `F`

Example 4. The action
`U/algebra/algebra.omdoc?group closure present O/omdoc/ascii.omdoc?ascii write group.txt`
writes the presentation of the closure of the theory of groups to the file `group.txt` using an ASCII-based style.
`U/algebra/algebra.omdoc?group?inv component type xml` writes the type type of `U/algebra/algebra.omdoc?g`
in XML to standard output.

5 The HTTP Server

Invocation The MMT HTTP server can be started by feeding `ombase.war` to a servlet container such as Tomcat or Jetty. A particularly easy way to start it is by executing

```
java -jar jetty-runner.jar ombase.war
```

Jetty-runner is provided in the main directory or available as part of Jetty.

The MMT server will execute the `file startup.mmt` from the current directory to initialize itself. In particular, this file should use the command `base` and `catalog` or `ombase`. If logging is switched on, log output will go to standard output.

GET Requests Currently only GET requests are implemented.

A GET request to the URI `scheme:authority/;?D?Q?R?A` is answered with the result of `D?Q?R A` where `A` must be an action taking MMT-URI and returning Non-MMT-Object. Spaces in `A` must be written as underscores, special characters in `A` must be %-encoded.

If the URI contains less than 3 ?, the missing components default to being empty.

Similarly, a GET request to the URI `scheme:authority/D?Q?R?A` is answered with the result of `D?Q?R A` (where `D?Q?R` is resolved relative to the base URI of the server set in the startup script).

Example 5. `http://localhost/D?Q?R?component_type_present_U` retrieves the type of `D?Q?R` rendered with style `U`.

References

[BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986, Internet Engineering Task Force (IETF), 2005.

[HIJ+11] F. Horozal, A. Iacob, C. Jucovschi, M. Kohlhase, and F. Rabe. Combining Source, Content, Presentation, Narration, and Relational Representation. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2011.

[KŞ06] M. Kohlhase and I. Şucan. A Search Engine for Mathematical Formulae. In T. Ida, J. Calmet, and D. Wang, editors, *Artificial Intelligence and Symbolic Computation*, pages 241–253. Springer, 2006.

[Rab09] F. Rabe. The MMT Language. 2009.