The MMT Manual

Florian Rabe
Jacobs University Bremen

March 20, 2013

Abstract

This is a permanently out-of-date collection of documentation meant to supplement the published accounts on the MMT language and system. It focuses on technical details and is mainly written as a reference, not as an introduction or tutorial.

1 XML Syntax and MMT URIs

The MMT language and its semantics have been described in depth in [RK13]. 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. 1.7.

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:

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

Namespaces N have no semantics and only serve to disambiguate toplevel declarations. Modules are any kind of named resource that introduces a scope within which other resources (i.e., symbols) are introduced such as signatures, theories, ontologies. Modules may be nested, in which case they have qualified names. Symbols are any kind of named atomic resource such as constants, functions, predicates, sorts, axioms, theorems. The names of symbols may be qualified as well, which MMT uses to form qualified names for symbols induced by named imports.

We will use Q and R to range over qualified names. In a URI $N[?Q[?R]]$, N, Q, and R are called the namespace, module name, and symbol name, respectively. A URI of this form is called **absolute** if N is an absolute URI. Otherwise, it is called **relative** or an *MMTURI* **reference**. Every absolute MMT URI has exactly one of the following three forms: N, $N?Q$, or $N?Q?R$ where N is an absolute URI. Every relative MMT URI has exactly one of the following seven forms: n, $n?q$, $n?q?r$, $?/q$, $?/q?r$, $??r$, or $??/r$ where n is a relative (possibly empty) URI.

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 resolution of an MMT URI reference u against an absolute base URI U is defined as follows:

1. u is of the form n, $n?q$, or $n?q?r$: u is resolved relative to the namespace N of U. (A possible module or symbol name in U are ignored.) If N' is the result of resolving n against N according to RFC 3986, then the resulting MMT URI is N', $N'?q$, or $N'?q?r$, respectively.
   Note that in the special case where n is empty, this implies $N' = n$. (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 in the same way as ..)

2. u is of the form $?/q$ or $?/q?r$: u is resolved relative to the namespace N and module name Q of U. (A possible symbol name in U is ignored. It is an error if U has no module name.) The resolution is $N?Q/q$ or $N?Q/q?r$, respectively.

3. u is of the form $??r$ or $??/r$ and $U = N?Q?R$. (It is an error if U is a module or document URI.) The resolution is $N?Q?r$ or $N?Q?R/r$, respectively.

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: *MMTURI*s that contain less than two occurrences of ?, can also be written with 2 $?s$ by appending ?. In other words, $n??$ and $N?Q?$ abbreviate N and $N?Q$, respectively.

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

Relationship with URIs Every absolute/relative *MMTURI* is also a legal absolute/relative URI. In particular, if we consider an *MMTURI* $N?Q?R$ as a URI, then $Q?R$ is its query component. Moreover, the *MMTURI* resolution of $n?q?r$ against $D?Q?R$ is identical to the usual resolution of relative URIs.

The situation is more complicated for those relative *MMTURI*s that begin with ?/ or ??. Here, the resolution must be implemented separately. This is unavoidable: In order to subsume common practices regarding XML namespaces, the module and symbol name must be put into the query component; but URIs do not permit relative resolution within the query component.

Relationship between OpenMath identifiers and *MMTURI*s. Every absolute *MMTURI* is a triple of namespace, module name, and symbol name. This corresponds directly to the cdbase-cd-name triple in OpenMath identifiers [BCC+04]. Note that OpenMath use the fragment component when forming URIs from OpenMath identifiers; *MMTURI*s avoid this because it would preclude efficient retrieval of individual symbols.

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.

Label: `dref` (a reference to an external document unit)

Attributes:

name	type	value
`target`	*MMTURI*	the referenced document

Children:

Label: `mref` (a reference to an external document unit)

Attributes:

name	type	value
`target`	*MMTURI*	the referenced module

Children:

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{object}`

Comment: Here, a symbol can be a constant or a structure.

Comment: Depending on the children, we speak of *declared* and *defined* theories. In the latter case, the definiens is a theory expression.

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* & symbol*) | definition{object}`

Comment: The symbols are assignments, i.e., their name must be the same as that of a corresponding symbol in the domain, their types are predetermined, and their definientia required.

Comment: Depending on the children, we speak of *declared* and *defined* views. In the latter case, the definiens is a morphism expression.

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 (includes and notations optionally so). If a symbol with name n occurs in a module with URI M, the symbol is addressable via the URI $M?n$.

Label: `include` (inclusion of a theory/view/style into the containing theory/view/styles)

Attributes:

name	type	value
`from`	*MMTURI*	the included module, relative to containing module

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: `alias{name} & type{object}? & definition{object}? & notation{notation}? & role{string}`

Comment: Constants can occur both in theories and views. In the latter case, their type is predetermined and must be omitted, and their definiens is required.

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* & symbol*) | definition{object}`

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

Comment: Depending on the children, we speak of *declared* and *defined* structures. In the latter case, the definiens is a morphism expression.

Every notation has a role. The permitted values of `role` are given in Sect. 1.7. 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 Object Level Elements

The type `objects` 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>

Module Expressions Objects may denote MMT theories and morphisms. Besides OMS elements referring to MMT theories, theory expressions can arise from, e.g., instantiation, union, and pushout. Besides OMS elements referring to MMT views and structures, morphism expressions can arise from, e.g., instantiation, identity, composition, unit, pushout. 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"/>
  μ1
  ...
  μn
</OMA>
```

Identity id_T:

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

Theories are encoded like links.

1.6 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="n1">
    O1
  </omdoc>
  ...
  <omdoc name="nr">
    Or
  </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.

1.7 Notations in Styles

1.7.1 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
$@(\omega_1, \ldots, \omega_n)$	`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)	$\omega_2 :: \omega_1 :: \omega_3 :: nil$
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. 1.7.2 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. 1.7.2 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
  - `application`: the path of the applied function,
  - `binding`: the path of the binder,
  - `attribution`: the path of the key,
  - `hidden`: none,
  - values and foreign objects: none
- For the roles `Toplevel` and `toplevel`, the path is the path of the toplevel expression.

1.7.2 Syntax and Semantics of 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. 1.7.

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.)

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.

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 `mai`
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.

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.

1.7.3 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

2 Text Syntax

2.1 Module Level

Intuitively, modules are the named declarations that introduce scopes containing other named declarations (other modules or symbols).

2.2 Symbol Level

Intuitively, symbols are the atomic named declarations contained in modules.

2.3 Object Level

Intuitively, objects are the unnamed components occurring in named declarations. This includes in particular the actual MMT objects that occur as the types and definientia. But for the purposes of the text syntax, it also all other components such as notations and roles.

2.3.1 Text Notations

Currently, the best available write-up is in [GIR13].

2.3.2 Pragmatic Syntax

Currently, the best available write-up is in [GIR13].

2.3.3 Parsing

Currently, the best available write-up is in [GIR13].

2.4 Structural Delimiters

Editor Support jEdit
VI

Font Support GNU unifont

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.

Archives can be manipulated via the API functions or via the shell (see Sect. 5.5).

Dimensions MMT supports the following dimensions, which occur as toplevel folders in an archive. The following dimensions contain the same tree of subfolders and files.

- `source`: source files in any language, in particular MMT text syntax
- `compiled`: the files in MMT XML syntax
- `narration`: similar to the ones in `compiled` but containing only the narrative outline, i.e., all MMT modules are replaced with `mref` elements
- `notation`: index of notations
- `relational`: relational index
- `mws`: MathWebSearch [KŞ06] index

The names of the folders `source` and `compiled` may be changed by using the `compilation` key in the manifest (see below).

The `content` dimension contains one file for every MMT module. The directory structure follows that of the MMT namespaces. In particular, the folder immediate below `content` is named by the scheme and the authority of the namespace URI. (`..` is used as a separator instead of `://` to avoid character restrictions of operating systems.) Below these folders, one subfolder is used for every path segment.

The `scala` dimension contains one file for every file in the `content` directory. It contains the Scala code extract from the MMT modules as described in [KMR13].

Manifest Archives must contain a file `META-INT/MANIFEST.MF`. Syntactically, this file consists of line-wise `key:value` pairs. Except for colons in keys, all characters are allowed. However, beginning and trailing whitespace in keys and values is ignored. Empty lines and lines starting with `//` are ignored. Unless otherwise mentioned, keys should occur only once and later occurrences overwrite previous values.

The following keys have a predefined meaning and are used by MMT:

- `id`: The identifier of the archive.
- `compilation`: A string of the form `format_1 @ folder_1 -> ... -> format_n @ folder_n`. This identifies a compilation chain used for the compilation of the archive. `format_i` defines the format of the files in `folder_i`. `folder_1` and `folder_n` identify the folders storing the `source` and `compiled` dimensions; consequently, `format_1` is the source format of the archive, and `format_n` should be `omdoc`.
- `narration-base`: This is a URI that gives the base URI for the `source` documents and all dimensions with the same folder structure. It also doubles as the URI of the whole archive.
- `source-base`: This is a URI that gives the base URI for all modules in the `source` documents.

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

compile `compiled` is generated from `source`.

content (indexing) `content`, `narration`, `relational`, and `notation` are generated generically from `compiled`.

extract (code generation) `scala` is generated from `content`.

mar (packaging) `id.mar` is generated by packaging all dimensions.

The compilation works as follows. According to the compilation chain defined in the manifest, compilers are called that read the files in `folder_i` and write files into `folder_i+1`. For each step in the chain, the appropriate compiler is chosen according to the `format_i`. The available compilers are provided as extensions (see Sect. 8).

4 Catalogs

All access to MMT knowledge items is done via URIs, not via URLs. Therefore, the *catalog* is necessary: It maintains a translation map from URIs to URLs. It is initially empty and built incrementally.

The most important kind of catalog entry is a *prefix*-entry: A catalog entry (i, l) where i is a URI and l is a URL has the effect that every MMT URI l/r is translated to i/r. Such entries are created automatically when registering an archive or explicitly via shell commands (see Sect. 5.7). In particular, if an archive is added, an entry (nb, loc) is generated that maps the `narration-base` of the archive to the physical location of its `narration` folder,

Moreover, for every archive, the URI of each module in its content folder to its physical location of the module.

5 The Shell Interface

The MMT shell is invoked by

```
java -cp scala/*;mmt-api.jar info.kwarc.mmt.api.RunNull
```

Further command line parameters are passed to the shell and executed as a command (see below). In particular,

```
java -cp mmt-api.jar info.kwarc.mmt.api.RunNull file F
```

can be used to execute a startup/configuration file `F`. After invocation, the shell can be controlled via STDIN/STDOUT.

The `-cp` parameter defines the Java classpath. Besides the main file `mmt-api.jar`, it must contain the jar file(s) of the Scala library [OSV07], which is here assumed to reside in a folder `scala`. Note that the classpath is a list of entries separated by `;` on Windows and `:` on Unix.

Additional classpath entries are necessary in the following cases:

- If the MMT web server is used, the classpath must contain `tiscaf.jar` [Gay08].
- If an MMT extension is used, it (and all its dependencies) must be in the classpath.

It is often reasonable to increase the memory available to MMT by using the appropriate Java parameter as in (here: 1024 MB)

```
java -Xmx1024m -cp scala/*;mmt-api.jar info.kwarc.mmt.api.RunNull
```

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

5.1 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.

5.2 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.

5.3 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.

5.4 Extension Commands

Extensions are registered using commands of the form `kind C ARGS`. `C` gives the qualified class name, and `ARGS` is a whitespace-separated list of arguments that is passed to the extension during initialization.

- `importer C ARGS`: This registers an importer. `C` must extend `info.kwarc.mmt.api.backend.Importer`. Importers are used to translate custom syntax into MMT. Relevant subinterfaces of `Importer` are
  - `info.kwarc.mmt.api.backend.Compiler` for compiling external files into MMT files
  - `info.kwarc.mmt.api.backend.QueryTransformer` for translating search queries into MMT objects

  Importers 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.
- `foundation C ARGS`: This registers a foundation in the sense of [RK13]. A foundation implements type and equality checking for a certain foundational theory.

5.5 Archive Commands

The following commands permit the registration and manipulation of archives:

- `archive add F`: This registers an archive with local root folder `F`.
- `archive ID O F`: This executes operation `O` on the archive with id `ID`, optionally restricted to the folder/file `F` (slashes as path separators). Legal values for `O` are `compile`, `content` (which will produce `content`, `narration`, `relational`, and `notation`), `mws`, `relational` (which reads the relational index), and `notation` (which reads the notation index).
- `archive mar F`: This builds the `mar` file and stores it in `F`.

5.6 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.

5.7 Catalog Commands

The following catalog commands can be used to add catalog entries explicitly:

- `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.

5.8 Server Commands

The MMT HTTP server (see Sect. 6) is controlled by the commands

- `server on P`: This starts the server on port `P`.
- `server off`: This shuts down the server.

6 The HTTP Interface

Besides being an API, the main interface for both humans and machines to the MMT system is via the web server. See Sect. 5.8 for how to start and stop the server.

It does not matter whether a request originates from the local or a remote machine.

6.1 Human Interface

The MMT web server can be accessed with any browser, e.g., by pointing it to `http://localhost:8080` after starting the server with `server on 8080` on the shell.

6.2 Machine Interface

The server responds to a number of requests that permit software systems to interact with MMT.

GET requests A GET request of the path `/:mmt?URI_A` is answered with the result of `URI A` where `A` is any action taking a MMT-URI. 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.

Example 5. `http://localhost:8080/:mmt?D?Q?R?component_type_present_U` retrieves the type of `D?Q?R` rendered with style `U`.

POST requests POST requests are used to access the query [Rab12] and the computation server.

7 The API

8 MMT Extensions and Plugins

MMT provides several interfaces for language-specific customization in external implementations. All extensions must provide a class `C` (as a qualified Java class name using dots to separate components), instantiate a certain interface, and have a constructor that takes no arguments.

Extensions are registered by giving the class name `C` (see Sect. 5.4). They are instantiated using Java reflection, and it is the user's responsibility to make sure that `C` is on the class path at the time of registration.

Extensions are initialized using an `init` method that takes a `List[String]` argument that is provided during registration.

References

[BCC+04] S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See http://www.openmath.org/standard/om20.

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

[Gay08] A. Gaydenko. tiscaf http server, 2008. http://gaydenko.com/scala/tiscaf/httpd/.

[GIR13] D. Ginev, M. Iancu, and F. Rabe. Integrating Content and Narration-Oriented Document Formats. see http://kwarc.info/frabe/Research/GIR_mmtlatex_13.pdf, 2013.

[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*, pages 212–227. Springer, 2011.

[KMR13] M. Kohlhase, F. Mance, and F. Rabe. A Universal Machine for Biform Theory Graphs. see http://kwarc.info/frabe/Research/KMR_uom_13.pdf, 2013.

[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.

[OSV07] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. artima, 2007.

[Rab12] F. Rabe. A Query Language for Formal Mathematical Libraries. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 142–157. Springer, 2012.

[RK13] F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 2013. conditionally accepted; see http://arxiv.org/abs/1105.0548.