An Infrastructure for Presenting Semantic Macros in sTeX*

Michael Kohlhase
Jacobs University, Bremen
http://kwarc.info/kohlhase

May 7, 2008

Abstract

The `presentation` packge is a central part of the sTeX collection, a version of TeX/LaTeX that allows to markup TeX/LaTeX documents semantically without leaving the document format, essentially turning TeX/LaTeX into a document format for mathematical knowledge management (MKM).

This package supplies an infrastructure that allows to specify the presentation of semantic macros, including preference-based bracket elision. This allows to markup the functional structure of mathematical formulae without having to lose high-quality human-oriented presentation in LaTeX. Moreover, the notation definitions can be used by MKM systems for added-value services, either directly from the sTeX sources, or after translation.

Contents

*Version v0.9e (last revised 2007/09/03)

1 Introduction

The `presentation` package supplies an infrastructure that allows to specify the presentation of semantic macros, including preference-based bracket elision. This allows to markup the functional structure of mathematical formulae without having to lose high-quality human-oriented presentation in LaTeX. Moreover, the notation definitions can be used by MKM systems for added-value services, either directly from the sTeX sources, or after translation.

sTeX is a version of TeX/LaTeX that allows to markup TeX/LaTeX documents semantically without leaving the document format, essentially turning TeX/LaTeX into a document format for mathematical knowledge management (MKM).

The setup for semantic macros described in the sTeX `modules` package works well for simple mathematical functions: we make use of the macro application syntax in TeX to express function application. For a simple function called "foo", we would just declare `\symdef{foo}[1]{foo(#1)}` and have the concise and intuitive syntax `\foo{x}` for $foo(x)$. But mathematical notation is much more varied and interesting than just this.

2 The User Interface

In this package we will follow the sTeX approach and assume that there are four basic types of mathematical expressions: symbols, variables, applications and binders. Presentation of the variables is relatively straightforward, so we will not concern ourselves with that. The application of functions in mathematics is mostly presented in the form $f(a_1, \ldots, a_n)$, where f is the function and the a_i are the arguments. However, many commonly-used functions from this presentational scheme: for instance binomial coefficients: $\binom{n}{k}$, pairs: $\langle a, b\rangle$, sets: $\{x \in S \mid x^2 \neq 0\}$, or even simple addition: $3 + 5 + 7$. Note that in all these cases, the presentation is determined by the (functional) head of the expression, so we will bind the presentational infrastructure to the operator.

2.1 Mixfix Notations

For the presentation of ordinary operators, we will follow the approach used by the Isabelle theorem prover. There, the presentation of an n-ary function (i.e. one that takes n arguments) is specified as $\langle pre\rangle\langle arg_0\rangle\langle mid_1\rangle\cdots\langle mid_n\rangle\langle arg_n\rangle\langle post\rangle$, where the $\langle arg_i\rangle$ are the arguments and $\langle pre\rangle$, $\langle post\rangle$, and the $\langle mid_i\rangle$ are presentational material. For instance, in infix operators like the binary subset operator, $\langle pre\rangle$ and $\langle post\rangle$ are empty, and $\langle mid_1\rangle$ is $\subseteq$. For the ternary conditional operator in a programming language, we might have the presentation pattern `if`$\langle arg_1\rangle$`then`$\langle arg_2\rangle$`else`$\langle arg_3\rangle$`fi` that utilizes all presentation positions.

`\mixfix*`

The `presentation` package provides mixfix declaration macros `\mixfixi`, `\mixfixii`, and `\mixfixiii` for unary, binary, and ternary functions. This covers most of the cases, larger arities would need a different argument pattern.[1] The

[1] If you really need larger arities, contact the author!

call pattern of these macros is just the presentation pattern above. In general, the mixfix declaration of arity i has $2n+1$ arguments, where the even-numbered ones are for the arguments of the functions and the odd-numbered ones are for presentation material. For instance, to define a semantic macro for the subset relation and the conditional, we would use the markup in Figure 1.

```
\symdef{sseteq}[2]{\mixfixii{}{#1}{\subseteq}{#2}{}}
\symdef{sseteq}[2]{\infix\subseteq{#1}{#2}}
\symdef{ite}[2]{\mixfixiii{{\tt{if}}\;}{#1}
                          {\;{\tt{then}}\;}{#2}
                          {\;{\tt{else}}\;}{#3}{\;{\tt{fi}}}}}
```

source	presentation
`\sseteq{S}T`	$(S \subseteq T)$
`\ite{x<0}{-x}x`	`if` $x < 0$ `then` $-x$ `else` x `fi`

Example 1: Declaration of mixfix operators

For certain common cases, the `presentation` package provides shortcuts
`\prefix` for the mixfix declarations. The `\prefix` macro allows to specify a prefix presentation for a function (the usual presentation in mathematics). Note that it is better to specify `\symdef{uminus}[1]{\prefix{-}{#1}}` than just `\symdef{uminus}[1]{-#1}`, since we can specify the bracketing behavior in the former (see Section 2.3).

`\postfix` The `\postfix` macro is similar, only that the function is presented after the argument as for e.g. the factorial function: 5! stands for the result of applying the factorial function to the number 5. Note that the function is still the first argument to the `\postfix` macro: we would specify the presentation for the factorial function with `\symdef{factorial}[1]{\postfix{!}{#1}}`.

`\infix` Finally, we provide the `\infix` macro for binary operators that are written between their arguments (see Figure 1).

2.2 n-ary Associative Operators

Take for instance the operator for set union: formally, it is a binary function on sets that is associative (i.e. $(S_1 \cup S_2) \cup S_3 = S_1 \cup (S_2 \cup S_3)$), therefore the brackets are often elided, and we write $S_1 \cup S_2 \cup S_3$ instead (once we have proven associativity). Some authors even go so far to introduce set union as a n-ary operator, i.e. a function that takes an arbitrary (positive) number of arguments. We will call such operators **n-ary associative**.

EdNote(1) Specifying the presentation[1] of n-ary associative operators in `\symdef` forms is not straightforward, so we provide some infrastructure for that. As we cannot predict the number of arguments for n-ary operators, we have to give them all at once, if we want to maintain our use of TEX macro application to specify

[1] EdNote: introduce the notion of presentation above

function application. So a semantic macro for an n-ary operator will be applied as `\nunion{`$\langle a_1\rangle$`,...,`$\langle a_n\rangle$`}`, where the sequence of n logical arguments $\langle a_i\rangle$ are supplied as one TEX argument which contains a comma-separated list. We provide variants of the mixfix declarations presented in section 2.1 which deal with associative arguments. For instance, the variant `\mixfixa` allows to specify n-ary associative operators. `\mixfixa{`$\langle pre\rangle$`}{`$\langle arg\rangle$`}{`$\langle post\rangle$`}{`$\langle op\rangle$`}` specifies a presentation, where $\langle arg\rangle$ is the associative argument and $\langle op\rangle$ is the corresponding operator that is mapped over the argument list; as above, $\langle pre\rangle$, $\langle post\rangle$, are prefix and postfix presentational material. For instance, the finite set constructor could be constructed as

\mixfixa

```
\newcommand{\fset}[1]{\mixfixa[p=0]{\{}{#1}{\}}{,}}
```

\assoc

The `\assoc` macro is a convenient abbreviation of a `\mixfixa` that can be used in cases, where $\langle pre\rangle$ and $\langle post\rangle$ are empty (i.e. in the majority of cases). It takes two arguments: the presentation of a binary operator, and a comma-separated list of arguments, it replaces the commas in the second argument with the operator in the first one. For instance `\assoc\cup{S_1,S_2,S_3}` will be formatted to $S_1 \cup S_2 \cup S_3$. Thus we can use `\def\nunion#1{\assoc\cup{#1}}` or even `\def\nunion{\assoc\cup}`, to define the n-ary operator for set union in TEX. For the definition of a semantic macro in STEX, we use the second form, since we are more conscious of the right number of arguments and would declare `\symdef{nunion}[1]{\assoc\cup{#1}}`.[2]

EdNote(2)

\prefixa
\postfixa

These macros `\prefix` and `\postfix` have n-ary variants `\prefixa` and `\postfixa` that take an arbitrary number of arguments (mathematically; syntactically grouped into one TEX argument). These take an extra separator argument.[3]

EdNote(3)

\mixfixia
\mixfixai
\mixfixaa

The `\mixfixii` macro has variants `\mixfixia`, `\mixfixai`, and `\mixfixaa`, which allow to make one or two arguments in a binary function associative[2]. A use case for the second macro is an nary function type operator `\fntype`, which can be defined via

```
\def\fntype#1#2{\mixfixai{}{#1}\rightarrow{#2}{}\times}
```

and which will format `\fntype{\alpha,\beta,\gamma}\delta` as $\alpha \times \beta \times \gamma \to \delta$.

2.3 Precedence-Based Bracket Elision

With the infrastructure supplied by the `\assoc` macro we could now try to combine set union and set intersection in one formula. Then, writing

```
\nunion{\ninters{a,b},\ninters{c,d}}        (1)
```

would yield $((a \cap b) \cup (c \cap d))$, and not $a \cap b \cup c \cap d$ as we would like, since $\cap$ binds stronger than $\cup$. Dropping outer brackets in the presentations of the presentation

[2] EdNote: think about big operators for ACI functions

[3] EdNote: think of a good example!

[2] If you really need larger arities with associative arguments, contact the package author!

of the operators will not help in general: it would give the desired form for (1) but $a \cap b \cup c \cap d$ for (2), where we would have liked $(a \cup b) \cap (c \cup d)$

```
\ninters{\nunion{a,b},\nunion{c,d}}
```
(2)

In mathematics, brackets are elided, whenever the author anticipates that the reader can understand the formula without them, and would be overwhelmed with them. To achieve this, there are set of common conventions that govern bracket elision. The most common is to assign precedences to all operators, and elide brackets, if the precedence of the operator is lower than that of the context it is presented in. In our example above, we would assign $\cap$ a lower precedence than $\cup$ (and both a lower precedence than the initial precedence). To compute the presentation of (2) we start out with the `\ninters`, elide its brackets (since the precedence n of $\cup$ is lower than the initial precedence i), and set the context precedence for the arguments to n. When we present the arguments, we present the brackets, since the precedence of `nunion` is lower than the context precedence n.

This algorithm, which we call **precedence-based bracket elision** goes a long way towards approximating mathematical practice. Note that full bracket elision in mathematical practice is a reader-oriented process, it cannot be fully mechanical, e.g. in $(a \cap b \cap c \cap d \cap e \cap f \cap g) \cup h$ we better put the brackets around the septary intersection to help the reader even thoug they could have been elided by our algorithm. Therefore, the author has to retain full control over bracketing in a bracket elision architecture (otherwise it would become impossible to explain the concept of associativity).[4].

EdNote(4)

Precedence	Operators	Comment
200	+,-	unary
200	^	exponentiation
400	$*, \wedge, \cap$	multiplicative
500	$+, -, \vee, \cup$	additive
600	/	fraction
700	$=, \neq, \leq, <, >, \geq$	relation

Figure 1: Common Operator Precedences

In SᴛEX we supply an optional keyval arguments to the mixfix declarations and their abbreviations that allow to specify precedences: The key `p` key is used to specify the **operator precedence**, and the keys `p`$\langle i \rangle$ can be used to specify the **argument precedences**. The latter will set the precedence level while processing the arguments, while the operator precedence invokes brackets, if it is larger than the current precedence level — which is set by the appropriate argument precedence by the dominating operators or the outer precedence.

`p`
`pi`
`pii`
`piii`

[4] EdNote: think about how to implement that

If none of the precedences is specified, then the defaults are assumed. The operator precedence is set to the default operator precedence, which defaults to 1000 and can be set by `\setDefaultPrecedence{`⟨*prec*⟩`}` where ⟨*prec*⟩ is an integer. The argument precedences default to the operator precedence.

\setDefaultPrecedence

Figure 1 gives an overview over commonly used precedences. Note that most operators have precedences lower than the default precedence of 1000, otherwise the brackets would not be elided. For our examples above, we would define

```
\newcommand{\nunion}[1]{\assoc[p=500]{\cup}{#1}}
\newcommand{\ninters}[1]{\assoc[p=400]{\cap}{#1}}
```

to get the desired behavior.

Note that the presentation macros uses round brackets for grouping by default. We can specify other brackets via two more keywords: `lbrack` and `rbrack`. Just as above, we can also reset the default brackets with `\setDefaultLeftBracket{`⟨*lb*⟩`}`and `\setDefaultRightBracket{`⟨*rb*⟩`}` where ⟨*lb*⟩ and ⟨*rb*⟩ expand to the desired brackets. Note that formula parts that look like brackets usually are not. For instance, we should not define the finite set constructor via

lbrack
rbrack
\setDefaultLeftBracket
\setDefaultRightBracket

```
\newcommand{\fset}[1]{\assoc[lbrack=\{,rbrack=\}]{,}{#1}}
```

where the curly braces are used as brackets, but as presented in section 2.2 even though both would format `\fset{a,b,c}` as $\{a, b, c\}$. In the encoding here, an operator with suitably high operator precedence would be able to make the brackets disappear.

2.4 Flexible Elision

There are several situations in which it is desirable to display only some parts of the presentation:

- We have alreday seen the case of redundant brackets above
- Arguments that are strictly necessary are omitted to simplify the notation, and the reader is trusted to fill them in from the context.
- Arguments are omitted because they have default values. For example $\log_{10} x$ is often written as $\log x$.
- Arguments whose values can be inferred from the other arguments are usually omitted. For example, matrix multiplication formally takes five arguments, namely the dimensions of the multiplied matrices and the matrices themselves, but only the latter two are displayed.

Typically, these elisions are confusing for readers who are getting acquainted with a topic, but become more and more helpful as the reader advances. For experienced readers more is elided to focus on relevant material, for beginners representations are more explicit. In the process of writing a mathematical document

for traditional (print) media, an author has to decide on the intended audience and design the level of elision (which need not be constant over the document though). With electronic media we have new possibilities: we can make elisions flexible. The author still chooses the elision level for the initial presentation, but the reader can adapt it to her level of competence and comfort, making details more or less explicit.

`\elide` To provide this functionality, the `presentation` package provides the `\elide` macro allows to asociate a text with an integer **visibility level** and group them into **elision groups**. High levels mean high elidability.

Elision can take various forms in print and digital media. In static media like traditional print on paper or the PostScript format, we have to fix the elision level, and can decide at presentation time which elidable tokens will be printed and which will not. In this case, the presentation algorithm will take visibility thresholds T_g for every elidability group g as a user parameter and then elide `\setelevel` (i.e. not print) all tokens in visibility group g with level $l > T_g$. We specify this threshold for via the `\setelevel` macro. For instance in the example below, we have a two type annotations `par` for type parameters and `typ` for type annotations themselves.

```
$\mathbf{I}\elide{par}{500}{^\alpha}\elide{typ}{100}{_{\alpha\to\alpha}}
    :=\lambda{X\elide{ty}{500}{_\alpha}}.X$
```

The visibility levels in the example encode how redundant the author thinks the elided parts of the formula are: low values show high redundancy. In our example the intuition is that the type paraemter on the **I** cominator and the type annotation on the bound variable X in the λ expression are of the same obviousness to the reader. So in a document that contains `\setegroup{typ}{1000}` and `\setegroup{an}{1000}` will show $\mathbf{I} := \lambda X.X$ eliding all redundant information. If we have both values at 400, then we will see $\mathbf{I}^\alpha := \lambda X_\alpha.X$ and only if the threshold for `typ` dips below 100, then we see the full information: $\mathbf{I}^\alpha_{\alpha\to\alpha} := \lambda X_\alpha.X$.

In an output format that is capable of interactively changing its appearance, e.g. dynamic XHTML+MathML (i.e. XHTML with embedded Presentation MathML formulas, which can be manipulated via JavaScript in browsers), an application can export the information about elision groups and levels to the target format, and can then dynamically change the visibility thresholds by user interaction. Here the visibility threshold would also be used, but here it only determines the default rendering; a user can then fine-tune the document dynamically to reveal elided material to support understanding or to elide more to increase conciseness.

The price the author has to pay for this enhanced user experience is that she has to specify elided parts of a formula that would have been left out in conventional LaTeX. Some of this can be alleviated by good coding practices. Let us consider the log base case. This is elided in mathematics, since the reader is expected to pick it up from context. Using semantic macros, we can mimic this behavior: defining two semantic macros: `\logC` which picks up the log base from the context

via the \logbase macro and \logB which takes it as a (first) argument.

```
\provideEdefault{logbase}{10}
\symdef{logB}[2]{\prefix{\mathrm{log}\elide{base}{100}{_{#1}}}{#2}}
\abbrdef{logC}[1]{\logB{\fromEcontext{logbase}}{#1}}
```

\provideEdefault Here we use the \provideEdefault macor to initialize a LaTeX token register for the logbase default, which we can pick up from the elision context using \fromEcontext in the definition of \logC. Thus \logC{x} would render as $\log_{10}(x)$ with a threshold of 50 for base and as $\log_2$, if the local TeX group e.g. given by the assertion environment contains a \setEdefault{logbase}{2}.

\fromEcontext

\setEdefault

2.5 Hyperlinking

EdNote(5) [5]

2.6 Variable Names

EdNote(6) [6]

\vname \vname identifies a token sequence as a name, and provides an ASCII (XML-compatible) identifier for it. The optional argument is the identifier, and the second one the LaTeX representation. The identifier can also be used with \vnameref for copy and paste.[7]

EdNote(7)

3 The Implementation

We first make sure that the KeyVal package is loaded (in the right version). For LaTeXML, we also initialize the package inclusions.

```
⟨package⟩\RequirePackage{keyval}[1997/11/10]
⟨*ltxml⟩
# -*- CPERL -*-
package LaTeXML::Package::Pool;
use strict;
use LaTeXML::Package;
RequirePackage('keyval');
⟨/ltxml⟩
```

We will first specify the default precedences and brackets, together with the macros that allow to set them.

```
⟨*package⟩
\def\pres@default@precedence{1000}
\def\setDefaultPrecedence#1{\def\pres@default@precedence{#1}}
\def\pres@initial@precedence{1000}
```

[5] EdNote: describe what we want to do here
[6] EdNote: what is the problem?
[7] EdNote: does this really work

```
\def\setInitialPrecedence#1{\def\pres@initial@precedence{#1}}
\def\pres@current@precedence{\pres@initial@precedence}
\def\pres@default@lbrack{(}\def\pres@lbrack{\pres@default@lbrack}
\def\pres@default@rbrack{)}\def\pres@rbrack{\pres@default@rbrack}
\def\setDefaultLeftBracket#1{\def\pres@default@lbrack{#1}}
\def\setDefaultRightBracket#1{\def\pres@default@rbrack{#1}}
⟨/package⟩
```

3.1 The System Commands

EdNote(8)

`\PrecSet` `\PrecSet` will set the default precedence.[8]

```
⟨package⟩\def\PrecSet#1{\def\pres@default@precedence{#1}}
⟨*ltxml⟩
⟨/ltxml⟩
```

`\PrecWrite` `\PrecWrite` will write a bracket, if the precedence mandates it, i.e. if `\pres@p` is greater than the current `\pres@current@precedence`

```
⟨package⟩\def\PrecWrite#1{\ifnum\pres@current@precedence>\pres@p\else{#1}\fi}
```

3.2 Mixfix Operators

```
⟨*package⟩
\def\clearkeys{\let\pres@p@key=\relax
\let\pres@pi@key=\relax%
\let\pres@pi@key=\relax%
\let\pres@pii@key=\relax%
\let\pres@piii@key=\relax}
\define@key{mi}{lbrack}{\def\pres@lbrack@key{#1}}
\define@key{mi}{rbrack}{\def\pres@lbrack@key{#1}}
\define@key{mi}{p}{\def\pres@p@key{#1}}
\define@key{mi}{pi}{\def\pres@pi@key{#1}}
\def\prep@keys@mi%
{\edef\pres@lbrack{\@ifundefined{pres@lbrack@key}{\pres@default@lbrack}{\pres@lbrack@key}}
\edef\pres@rbrack{\@ifundefined{pres@rbrack@key}{\pres@default@rbrack}{\pres@rbrack@key}}
\edef\pres@p{\@ifundefined{pres@p@key}{\pres@default@precedence}{\pres@p@key}}
\edef\pres@pi{\@ifundefined{pres@pi@key}{\pres@p}{\pres@pi@key}}}
⟨/package⟩
⟨*ltxml⟩
DefKeyVal('mi','lbrack','Semiverbatim');
DefKeyVal('mi','rbrack','Semiverbatim');
DefKeyVal('mi','p','Semiverbatim');
DefKeyVal('mi','pi','Semiverbatim');
⟨/ltxml⟩
```

`\mixfixi`

```
⟨*package⟩
\newcommand{\mixfixi}[4][]%key, pre, arg, post
```

[8] EdNote: need to implement this in LaTeXML?

```
{\setkeys{mi}{#1}\prep@keys@mi\clearkeys
\PrecWrite\pres@lbrack% write bracket if necessary
#2{\edef\pres@current@precedence{\pres@pi}#3}#4%
\PrecWrite\pres@rbrack}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\mixfixi OptionalKeyVals:mi {}{}{}',
                "<omdoc:prototype>"
       .  "<om:OMA>"
       .    "<om:OMS cd='' name=''/>"##### need to get $cd and $name here.
       .    "<omdoc:expr name='arg'/>"
       .  "</om:OMA>"
            ."</omdoc:prototype>"
            ."<omdoc:rendering ?&KeyVal(#1,'p')(precedence='&KeyVal(#1,'p')')>"
       .  "<m:mrow>"
       .    "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
       .    "<omdoc:render name='arg' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
       .    "<ltx:Math><ltx:XMath>#4</ltx:XMath></ltx:Math>"
       .  "</m:mrow>"
            ."</omdoc:rendering>",
      mode=>'inline_math');
⟨/ltxml⟩
```

\mixfixa

```
⟨*package⟩
\newcommand{\mixfixa}[5][]%key, pre, arg, post, assocop
{\setkeys{mi}{#1}\prep@keys@mi\clearkeys%
\PrecWrite\pres@lbrack{#2}{\@assoc\pres@pi{#5}{#3}}{#4}\PrecWrite\pres@rbrack}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\mixfixa OptionalKeyVals:mi {}{}{}{}',
                "<omdoc:prototype>"
       .  "<om:OMA>"
       .    "<om:OMS cd='' name=''/>"##### need to get $cd and $name here.
       .    "<omdoc:exprlist name='args'>"
       .      "<omdoc:expr name='arg'/>"
       .    "</omdoc:exprlist>"
       .  "</om:OMA>"
            ."</omdoc:prototype>"
            ."<omdoc:rendering ?&KeyVal(#1,'p')(precedence='&KeyVal(#1,'p')')>"
       .  "<m:mrow>"
       .    "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
       .    "<omdoc:iterate name='args' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
            .      "<omdoc:separator>"
       .       "<ltx:Math><ltx:XMath>#5</ltx:XMath></ltx:Math>"
       .      "</omdoc:separator>"
       .      "<omdoc:render name='arg' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
            .    "</omdoc:iterate>"
       .    "<ltx:Math><ltx:XMath>#4</ltx:XMath></ltx:Math>"
       .  "</m:mrow>"
```

```
                ."</omdoc:rendering>",
        mode=>'inline_math');
⟨/ltxml⟩

⟨*package⟩
\define@key{mii}{lbrack}{\def\pres@lbrack@key{#1}}
\define@key{mii}{rbrack}{\def\pres@lbrack@key{#1}}
\define@key{mii}{p}{\def\pres@p@key{#1}}
\define@key{mii}{pi}{\def\pres@pi@key{#1}}
\define@key{mii}{pii}{\def\pres@pii@key{#1}}
\def\prep@keys@mii{\prep@keys@mi%
\edef\pres@pii{\@ifundefined{pres@pii@key}{\pres@p}{\pres@pii@key}}%
\let\pres@pii@key=\relax}
⟨/package⟩
⟨*ltxml⟩
DefKeyVal('mii','lbrack','Semiverbatim');
DefKeyVal('mii','rbrack','Semiverbatim');
DefKeyVal('mii','p','Semiverbatim');
DefKeyVal('mii','pi','Semiverbatim');
DefKeyVal('mii','pii','Semiverbatim');
⟨/ltxml⟩
```

\mixfixii

```
⟨*package⟩
\newcommand{\mixfixii}[6][]%key, pre, arg1, mid, arg2, post
{\setkeys{mii}{#1}\prep@keys@mii\clearkeys%
\PrecWrite\pres@lbrack% write bracket if necessary
#2{\edef\pres@current@precedence{\pres@pi}#3}%
#4{\edef\pres@current@precedence{\pres@pii}#5}#6%
\PrecWrite\pres@rbrack}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\mixfixii OptionalKeyVals:mi {}{}{}{}{}',
                "<omdoc:prototype>"
       .  "<om:OMA>"
       .    "<om:OMS cd='' name=''/>"##### need to get $cd and $name here.
       .    "<omdoc:expr name='arg1'/>"
       .    "<omdoc:expr name='arg2'/>"
       .  "</om:OMA>"
            ."</omdoc:prototype>"
            ."<omdoc:rendering ?&KeyVal(#1,'p')(precedence='&KeyVal(#1,'p')')>"
       .  "<m:mrow>"
       .    "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
       .    "<omdoc:render name='arg1' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
       .    "<ltx:Math><ltx:XMath>#4</ltx:XMath></ltx:Math>"
       .    "<omdoc:render name='arg2' ?&KeyVal(#1,'pii')(precedence='&KeyVal(#1,'pii')')/>"
       .    "<ltx:Math><ltx:XMath>#6</ltx:XMath></ltx:Math>"
       .  "</m:mrow>"
            ."</omdoc:rendering>",
       mode=>'inline_math');
```

```
⟨/ltxml⟩
```

\mixfixia

```
⟨*package⟩
\newcommand{\mixfixia}[7][]%key, pre, arg1, mid, arg2, post, assocop
{\setkeys{mii}{#1}\prep@keys@mii\clearkeys%
\PrecWrite\pres@lbrack% write bracket if necessary
#2{\edef\pres@current@precedence{\pres@pi}#3}%
#4{\@assoc\pres@pii{#7}{#5}}#6%
\PrecWrite\pres@rbrack}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\mixfixia OptionalKeyVals:mi {}{}{}{}{}{}',
                "<omdoc:prototype>"
       .  "<om:OMA>"
       .    "<om:OMS cd='' name=''/>"##### need to get $cd and $name here.
       .    "<omdoc:expr name='arg1'/>"
       .    "<omdoc:exprlist name='args'>"
       .      "<omdoc:expr name='arg'/>"
       .    "</omdoc:exprlist>"
       .  "</om:OMA>"
            ."</omdoc:prototype>"
            ."<omdoc:rendering ?&KeyVal(#1,'p')(precedence='&KeyVal(#1,'p')')>"
       .  "<m:mrow>"
       .    "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
       .    "<omdoc:render name='arg1' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
       .    "<ltx:Math><ltx:XMath>#4</ltx:XMath></ltx:Math>"
       .    "<omdoc:iterate name='args' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
            .      "<omdoc:separator>"
       .      "<ltx:Math><ltx:XMath>#7</ltx:XMath></ltx:Math>"
       .      "</omdoc:separator>"
       .      "<omdoc:render name='arg' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
            .    "</omdoc:iterate>"
       .    "<ltx:Math><ltx:XMath>#6</ltx:XMath></ltx:Math>"
       .  "</m:mrow>"
            ."</omdoc:rendering>",
      mode=>'inline_math');
⟨/ltxml⟩
```

\mixfixai

```
⟨*package⟩
\newcommand{\mixfixai}[7][]%key, pre, arg1, mid, arg2, post, assocop
{\setkeys{mii}{#1}\prep@keys@mii\clearkeys%
\PrecWrite\pres@lbrack% write bracket if necessary
#2{\@assoc\pres@pi{#7}{#3}}%
#4{\edef\pres@current@precedence{\pres@pii}#5}#6%
\PrecWrite\pres@rbrack}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\mixfixai OptionalKeyVals:mi {}{}{}{}{}{}',
```

```
                "<omdoc:prototype>"
        .  "<om:OMA>"
        .    "<om:OMS cd='' name=''/>"##### need to get $cd and $name here.
        .    "<omdoc:exprlist name='args'>"
        .      "<omdoc:expr name='arg'/>"
        .    "</omdoc:exprlist>"
        .    "<omdoc:expr name='arg2'/>"
        .  "</om:OMA>"
                ."</omdoc:prototype>"
                ."<omdoc:rendering ?&KeyVal(#1,'p')(precedence='&KeyVal(#1,'p')')>"
        .  "<m:mrow>"
        .    "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
        .    "<omdoc:iterate name='args' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
                .      "<omdoc:separator>"
        .        "<ltx:Math><ltx:XMath>#7</ltx:XMath></ltx:Math>"
        .      "</omdoc:separator>"
        .      "<omdoc:render name='arg' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
                .    "</omdoc:iterate>"
        .    "<ltx:Math><ltx:XMath>#4</ltx:XMath></ltx:Math>"
        .    "<omdoc:render name='arg2' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
        .    "<ltx:Math><ltx:XMath>#6</ltx:XMath></ltx:Math>"
        .  "</m:mrow>"
                ."</omdoc:rendering>",
       mode=>'inline_math');
⟨/ltxml⟩

⟨*package⟩
\define@key{miii}{lbrack}{\def\pres@lbrack@key{#1}}
\define@key{miii}{rbrack}{\def\pres@lbrack@key{#1}}
\define@key{miii}{p}{\def\pres@p@key{#1}}
\define@key{miii}{pi}{\def\pres@pi@key{#1}}
\define@key{miii}{pii}{\def\pres@pii@key{#1}}
\define@key{miii}{piii}{\def\pres@piii@key{#1}}
\def\prep@keys@miii{\prep@keys@mii\edef\pres@piii{\@ifundefined{pres@piii@key}{\pres@p}{\pres@p
⟨/package⟩
⟨*ltxml⟩
DefKeyVal('miii','lbrack','Semiverbatim');
DefKeyVal('miii','rbrack','Semiverbatim');
DefKeyVal('miii','p','Semiverbatim');
DefKeyVal('miii','pi','Semiverbatim');
DefKeyVal('miii','pii','Semiverbatim');
DefKeyVal('miii','piii','Semiverbatim');
⟨/ltxml⟩
```

`\mixfixiii`

```
⟨*package⟩
\newcommand{\mixfixiii}[8][]%key, pre, arg1, mid1, arg2, mid2, arg3, post
{\setkeys{miii}{#1}\prep@keys@miii\clearkeys%
\PrecWrite\pres@lbrack% write bracket if necessary
#2{\edef\pres@current@precedence{\pres@pi}#3}%
```

```
#4{\edef\pres@current@precedence{\pres@pii}#5}%
#6{\edef\pres@current@precedence{\pres@pii}#7}#8%
\PrecWrite\pres@rbrack}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\mixfixiii OptionalKeyVals:mi {}{}{}{}{}{}{}',
                "<omdoc:prototype>"
       .  "<om:OMA>"
       .    "<om:OMS cd='' name=''/>"##### need to get $cd and $name here.
       .    "<omdoc:expr name='arg1'/>"
       .    "<omdoc:expr name='arg2'/>"
       .    "<omdoc:expr name='arg3'/>"
       .  "</om:OMA>"
            ."</omdoc:prototype>"
            ."<omdoc:rendering ?&KeyVal(#1,'p')(precedence='&KeyVal(#1,'p')')>"
       .  "<m:mrow>"
       .    "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
       .    "<omdoc:render name='arg1' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/>"
       .    "<ltx:Math><ltx:XMath>#4</ltx:XMath></ltx:Math>"
       .    "<omdoc:render name='arg2' ?&KeyVal(#1,'pii')(precedence='&KeyVal(#1,'pii')')/>"
       .    "<ltx:Math><ltx:XMath>#6</ltx:XMath></ltx:Math>"
       .    "<omdoc:render name='arg3' ?&KeyVal(#1,'piii')(precedence='&KeyVal(#1,'piii')')/>"
       .    "<ltx:Math><ltx:XMath>#8</ltx:XMath></ltx:Math>"
       .  "</m:mrow>"
            ."</omdoc:rendering>",
      mode=>'inline_math');
⟨/ltxml⟩
```

EdNote(9) `\prefix, \postfix`

`\prefix`, `\prefixa`, `\postfix` and `\postfixa`[9] are simple special cases of `\mixfixi` and `\mixfixa`.

```
⟨*package⟩
\newcommand{\prefix}[3][]%key, fn, arg
{\setkeys{mi}{#1}\prep@keys@mi\clearkeys
#2\PrecWrite\pres@lbrack% write bracket if necessary
{\edef\pres@current@precedence{\pres@pi}#3}%
\PrecWrite\pres@rbrack}
\newcommand{\postfix}[3][]%key, fn, arg
{\setkeys{mi}{#1}\prep@keys@mi\clearkeys
\PrecWrite\pres@lbrack% write bracket if necessary
{\edef\pres@current@precedence{\pres@pi}#3}%
\PrecWrite\pres@rbrack{#2}}
\newcommand{\prefixa}[4][]{\mixfixa[#1]{#2}{#3}{}{#4}}
\newcommand{\postfixa}[4][]{{#1}\mixfixa[#1]{}{#3}{#2}{#4}}
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\prefix OptionalKeyVals:mi {}{}',
                "<omdoc:prototype>"
               .  "<om:OMA>"
```

[9] EdNote: need prefixl and postfixl as well, use counters for precedences here.

```
                .    "<om:OMS cd='' name=''/>"##### need to get $cd and $name here.
                .    "<omdoc:expr name='arg1'/>"
                .  "</om:OMA>"
                ."</omdoc:prototype>"
                ."<omdoc:rendering ?&KeyVal(#1,'p')(precedence='&KeyVal(#1,'p')')>"
                .  "<m:mrow>"
                .    "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
                .    "<omdoc:render name='arg' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/
                .  "</m:mrow>"
                ."</omdoc:rendering>",
        mode=>'inline_math');
DefConstructor('\postfix OptionalKeyVals:mi {}{}',
                 "<omdoc:prototype>"
                .  "<om:OMA>"
                .    "<om:OMS cd='' name=''/>"##### need to get $cd and $name here.
                .    "<omdoc:expr name='arg1'/>"
                .  "</om:OMA>"
                ."</omdoc:prototype>"
                ."<omdoc:rendering ?&KeyVal(#1,'p')(precedence='&KeyVal(#1,'p')')>"
                .  "<m:mrow>"
                .    "<omdoc:render name='arg' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')')/
                .    "<ltx:Math><ltx:XMath>#2</ltx:XMath></ltx:Math>"
                .  "</m:mrow>"
                ."</omdoc:rendering>",
        mode=>'inline_math');
⟨/ltxml⟩
```

EdNote(10)

`\infix` `\infix`[10] is a simple special case of `\mixfixii`.

```
⟨*package⟩
\newcommand{\infix}[4][]{\mixfixii[#1]{}{#3}{#2}{#4}{}}
⟨/package⟩
⟨*ltxml⟩
DefMacro('\infix []{}{}{}','\mixfixii[#1]{}{#3}{#2}{#4}{}');
⟨/ltxml⟩
```

3.3 Associative Operators

`\@assoc` We are using functionality from the LaTeX core packages here to iterate over the arguments.

```
⟨*package⟩
\def\@assoc#1#2#3{% precedence, function, argv
\let\@tmpop=\relax% do not print the function the first time round
\@for\@I:=#3\do{\@tmpop% print the function
% write the i-th argument with locally updated precedence
{\edef\pres@current@precedence{#1}\@I}%
\let\@tmpop=#2}}%update the function
⟨/package⟩
```

[10] EdNote: need infixl as well, use counters for precedences here.

\assoc With the internal macro above, associatifivity is easily specified.

```
⟨package⟩\newcommand{\assoc}[3][]{\mixfixa[#1]{}{#3}{}{#2}}
⟨*ltxml⟩
DefConstructor('\assoc OptionalKeyVals:mi {}{}',
                "<omdoc:prototype>"
               .  "<om:OMA>"
               .    "<om:OMS cd='' name=''/>"##### need to get $cd and $name here.
               .    "<omdoc:exprlist name='args'>"
               .      "<omdoc:expr name='arg'/>"
               .    "</omdoc:exprlist>"
               .  "</om:OMA>"
               ."</omdoc:prototype>"
               ."<omdoc:rendering ?&KeyVal(#1,'p')(precedence='&KeyVal(#1,'p')')>"
               .  "<m:mrow>"
               .    "<omdoc:iterate name='args' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')'
               .      "<omdoc:separator>"
               .        "<ltx:Math><ltx:XMath>#3</ltx:XMath></ltx:Math>"
               .      "</omdoc:separator>"
               .      "<omdoc:render name='arg' ?&KeyVal(#1,'pi')(precedence='&KeyVal(#1,'pi')'
               .    "</omdoc:iterate>"
               .  "</m:mrow>"
               ."</omdoc:rendering>",
       mode=>'inline_math');
⟨/ltxml⟩
```

3.4 General Elision

\setegroup The elision macros are quite simple, a group `foo` is internally represented by a macro `foo@egroup`, which we set by a `\gdef`.

```
⟨package⟩\def\setegroup#1#2{\expandafter\def\csname #1@egroup\endcsname{#2}}
⟨*ltxml⟩
⟨/ltxml⟩
```

\setegroup Then the elision command is picks up on this (flags an error) if the internal macro does not exist and prints the third argument, if the elision value threshold is above the elision group threshold in the paper.

```
⟨*package⟩
\def\elide#1#2#3{\@ifundefined{#1@egroup}%
{\def\@elevel{1000}
\PackageError{presentation}{undefined egroup #1, assuming value 1000}%
{When calling \protect\elide{#1}... the elision group #1 has be have\MessageBreak
been set by \protect\setegroup before, e.g. by \protect\setegroup{an}{1000}.}}%
{\edef\@elevel{\csname #1@egroup\endcsname}}%
\ifnum\@elevel>#2\else{#3}\fi}
⟨/package⟩
⟨*ltxml⟩
⟨/ltxml⟩
```

\provideEdefault The \provideEdefault macro sets up the context for an elision default by locally defining the internal macro ⟨*default*⟩@edefault and (if necessary) exporting it from the module.

```
⟨*package⟩
\def\provideEdefault#1#2{\expandafter\def\csname#1@edefault\endcsname{#2}
\@ifundefined{this@module}{}%
{\expandafter\g@addto@macro\this@module{\expandafter\def\csname#1@edefault\endcsname{#2}}}}
⟨/package⟩
⟨*ltxml⟩
⟨/ltxml⟩
```

\setEdefault The \setEdefault macro just redefines the internal ⟨*default*⟩@edefault in the local group

```
⟨package⟩\def\setEdefault#1#2{\expandafter\def\csname #1@edfault\endcsname{#2}}
⟨*ltxml⟩
⟨/ltxml⟩
```

\fromEcontext The \fromEcontext macro just calls internal ⟨*default*⟩@edefault macro.

```
⟨package⟩\def\fromEcontext#1{\csname #1@edefault\endcsname}
⟨*ltxml⟩
⟨/ltxml⟩
```

3.5 Variable Names

EdNote(11) EdNote(12) \vname a name macro[11][12]

```
⟨*package⟩
\def\MOD@namedef#1{\expandafter\def\csname MOD@name@#1\endcsname}
\def\MOD@name[#1]#2{#2\def\@test{#2}\ifx\@test\empty\else\MOD@namedef{#1}{#2}\fi}
\def\vname{\@ifnextchar[\MOD@name{\MOD@name[]}}
⟨/package⟩
⟨*ltxml⟩
⟨/ltxml⟩
```

\vnameref

```
⟨package⟩\def\vnref#1{\csname MOD@name@#1\endcsname}
```

3.6 Hyperlinking

EdNote(13) this only works for internal links[13]

```
⟨package⟩\def\hrcr#1#2{\hyperlink{#1@\mod@id}{#2}}
⟨*ltxml⟩
⟨/ltxml⟩
```

[11] EdNote: add some documentation here
[12] EdNote: maybe this should go into the structuresharing package?
[13] EdNote: actually not at all!