`presentation.sty`: An Infrastructure for Presenting Semantic Macros in sTeX*

Michael Kohlhase & Deyan Ginev
Jacobs University, Bremen
`http://kwarc.info/kohlhase`

August 12, 2015

Abstract

The `presentation` package 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.

*Version v1.0 (last revised 2013/01/04)

Contents

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: $\{x \in S \,|\, x^2 \neq 0\}$, 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 Prefix & Postfix Notations

`\prefix`

The default notation for an object that is obtained by applying a function f to arguments a_1 to a_n is $f(a_1, \ldots, a_n)$. 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 2.4).

`\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}}`.

`\prefixa` `\postfixa`

`\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.[1]

EdN:1

Note that in STEX the `\prefix` and `\postfix` macros should primarily be used in `\symdef` declarations. For marking up applications of symbolic functions in text we should use the `\symdef`-defined semantic macros direct. For applications of function variables we have two options:

i) direct prefix markup of the form `f(x)`, where we have declared the symbol `f` to be a function via the `function` key of the enclosing environment — e.g. `omtext` (see [**Kohlhase:smmtf*:svn**]).

`\funapp` *ii)* using the `\funapp` macro as in `\funapp{f}{x}`, which leads to the same effect and is more general (e.g. for complex function variables, such as f_1'). Note that the default prefix rendering of the function is sufficient here, since we can otherwise make use of a user-defined application operator.

2.2 Mixfix Notations

For the presentation of more complex 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 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.

`\infix` For certain common cases, the `presentation` package provides shortcuts for the mixfix declarations. For instance, we provide the `\infix` macro for binary operators that are written between their arguments (see Figure 1).[2]

EdN:2

2.3 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

[1] EdNote: think of a good example!

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

[2] EdNote: really?

```
\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

operator, i.e. a function that takes an arbitrary (positive) number of arguments. We will call such operators n**-ary associative**.

EdN:3 Specifying the presentation[3] 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 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 2.2 which deal with

`\mixfixa` 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

```
\newcommand\fset[1]{\mixfixa{\{}{#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

EdN:4 `\symdef{nunion}[1]{\assoc\cup{#1}}`.[4]

`\mixfixia` `\mixfixai` The `\mixfixii` macro has variants `\mixfixia` and `\mixfixai` which allow to make one or two arguments in a binary function associative. A use case for the

[3] EdNote: introduce the notion of presentation above

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

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 $(\alpha \times \beta \times \gamma \to \delta)$

Finally, the `\mixfixiii` macro has the variants `\mixfixaii`, `\mixfixiai`, and `\mixfixiia` as above[2]. For instance we can use the first variant for a typing judgment using

```
\def\typej#1#2#3{\mixfixaii{}{#1}{\vdash_{\Sigma}}{#2}\colon{#3}{}{,}}
```

which formats `\typej{\Gamma,[x:\alpha],[y:\beta]}{f(x,y)}{\beta}` as

$$(\Gamma, [x : \alpha], [y : \beta] \vdash_\Sigma f(x, y) \colon \beta).$$

2.4 Precedence-Based Bracket Elision

In the infrastructure discussed above, we have completely ignored the fact that we use brackets to disambiguate the formula structure. The general baseline rule here is that we enclose any presented subformula with (round) brackets to mark it as a logical unit. If we applied this to the following formula that combines set union and set intersection

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

this would yield $((a \cap b) \cup (c \cap d))$, and not $a \cap b \cup c \cap d$ as we are used to. 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 — "$\cap$ binds stronger than $\cup$" in (1). The most common is to assign precedences to all operators, and elide brackets, if the precedence of the operator is larger than that of the context it is presented in (or equivalently: we only write brackets, if the operator precedence is smaller or equal to the context precedence). Note that this is more selective that simply dropping outer brackets which would yield $a \cap b \cup c \cap d$ for (2), where we would have liked $(a \cup b) \cap (c \cup d)$

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

In our example above, we would assign $\cap$ a larger precedence than $\cup$ (and both a larger precedence than the initial precedence to avoid outer brackets). To compute the presentation of (2) we start out with the `\ninters`, elide its brackets (since the precedence n of $\cup$ is larger than the initial precedence i), and set the context precedence for the arguments to n. When we present the arguments, we present

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

the brackets, since the precedence of `nunion` is larger 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 though they could have been elided by our algorithm. Therefore, the author has to retain full control[5] over bracketing in a bracket elision architecture. Otherwise it would become impossible to explain the concept of associativity in $(a \circ b) \circ c = a \circ (b \circ c)$, where we need the brackets for this one time on an otherwise associative operation $\circ$.

EdN:5

Precedence	Operators	Comment
800	+,-	unary
800	ˆ	exponentiation
600	$*, \wedge, \cap$	multiplicative
500	$+, -, \vee, \cup$	additive
400	/	fraction
300	$=, \neq, \leq, <, >, \geq$	relation

Figure 1: Common Operator Precedences

`p` `pi` `pii` `piii` `\iprec` `\niprec`

Furthermore, 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 smaller than the current precedence level — which is set by the appropriate argument precedence by the dominating operators or the outer precedence. The values of the precedence keys can be integers or `\iprec` for the infinitely large precedence or `\niprec` for the infinitely small precedence.

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 0. The argument precedences default to the operator precedence.

Figure 1 gives an overview over commonly used precedences. Note that most operators have precedences higher than the default precedence of 0, 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=600]{\cap}{#1}}
```

to get the desired behavior.

`lbrack` `rbrack`

Note that the presentation macros uses round brackets for grouping by default. We can specify other brackets via two more keywords: `lbrack` and `rbrack`.

[5] EdNote: think about how to implement that. We need a way to override precedences locally

Note that formula parts that look like brackets usually are not. For instance, we should not define the finite set constructor via

```
\newcommand\fset[1]{\assoc[lbrack=\{,rbrack=\}]{,}{#1}}
```
(3)

where the curly braces are used as brackets, but as presented in section 2.3 even though both would format `\fset{a,b,c}` as $\{a, b, c\}$. In the encoding here, an operator with suitably high operator precedence (it is the best practice u)would be able to make the brackets disappear. Thus the correct version of (3) is

```
\newcommand\fset[1]{\mixfixa[p=\iprec,pi=0]{\{}{#1}{\}}{,}}
```
(4)

Note that `\prefix` and `\postfix` and their variants declared in section 2.1 have brackets that do not participate (actively) in the precedence-based elision: function application brackets are not subject to elision. But the operator precedence `p` is still taken into account for outer brackets. The argument precedence `pi` has negative infinity as a default to avoid spurious brackets for arguments.

There is another use case for the `\mixfixi` macro that is not apparent at first glance. In some cases, we would naively construct presentations without a mixfix declaration, e.g.

```
\newcommand\half[1]{\frac{#1}2}
```
(5)

The the problem here is that the fraction does not participate in the precedence-based bracketing system, and in particular, the numerator will often have too many brackets (the incoming precedence is just passe through the `\half` macro). A better way is to wrap the intended presentation in a (somewhat spurious) `\mixfixi`, which we give the precedence `nobrackets`, which suppresses all (outer and argument) brackets for one level:

```
\newcommand\half[1]{\mixfixi[nobrackets]{}{\frac{#1}2}{}}
```
(6)

2.5 Flexible Elision

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

- We have already 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 associate 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 (i.e. not print) all tokens in visibility group g with level $l > T_g$. We specify this `\setegroup` threshold for via the `\setegroup` macro. For instance in the example below, we have a two type annotations `par` for type parameters and `typ` for type annotations themselves.

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

Example 2: Elision with Elision Groups

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 parameter on the $\mathbf{I}$ combinator 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}{0}` and `\setegroup{par}{0}` Figure 2 will show $\mathbf{I} := \lambda X.X$ eliding all redundant information. If we have both values at 600, then we will see $\mathbf{I}^\alpha := \lambda X_\alpha.X$ and only if the threshold for `typ` rises above 900, 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` macro 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 `setEdefault` the `assertion` environment contains a `\setEdefault{logbase}{2}`.

2.6 Other Layout Primitives

Not all mathematical layouts are producible with mixfix notations. A prime example are grid layouts which are marked up using the `array` element in TeX/LaTeX, e.g. for definition by cases as the (somewhat contrived) definition of the absolute value function in the upper part of Figure 3. We will now motivate the need of special layout primitives with this example. But this does not work for content

$$|x|:=\begin{cases} x & \text{if } x>0\\ -x & \text{if } x<0\\ 0 & \text{else}\end{cases}$$

```
|x|\colon=\left\{
\begin{array}{rl}
 x & x>0\\
-x & x<0\\
 0 & \text{else}
\end{array}
\right.
```

```
\symdef{piece}[2]{\parrayline{\parraycell{#1}}{\text{if}\;#2}}
\symdef{otherwise}[1]{\parrayline{\parraycell{#1}}{\text{else}}}
\symdef{piecewise}[1]{\left\{\begin{array}{rl}#1\end{array}\right.}
$|x|\colon=\piecewise{\piece{x}{x>0}\piece{-x}{x<0}\otherwise{0}}$
```

Example 3: A piecewise definition of the absolute value function

markup via semantic macros [**KohAmb:smmssl:ctan**], which wants to group formula parts by function. For definition by cases, we may want to follow the OpenMath `piece1` content dictionary [**CD:piece1:on**], which groups "piecewise" definitions into a constructor `piecewise`, whose children are a list of `piece` constructors optionally followed by an `otherwise`. If we want to mimic this by semantic macros in STeX (these are defined via `\symdef`; see [**KohAmb:smmssl:ctan**

] for details), we would naturally define `\piecewise` by wrapping an `array` environment (see the last line in Figure 3). Then we would naturally be tempted to define `\piece` via `\symdef{piece}[2]{#1&\text{if}\;{#2}\\}` and `\otherwise` via `\symdef{otherwise}[1]{#1&\text{else}}`. But this does not support the generation of separate notation definitions for `\piece` and `\otherwise`: here LaTeXMLhas to generate presentational information outside of the `array` context that provides the `&` and `\\` command sequences[3]. Therefore the `presentation` package provides the macros `\parrayline` and `\parraycell` that refactor this functionality.

`\parrayline` `\parrayline{`⟨*cells*⟩`}{`⟨*cell*⟩`}` is LaTeX-equivalent to ⟨*cells*⟩`&`⟨*cell*⟩`\\` and can thus be used to create array lines with one or more array cells: ⟨*cell*⟩ is the last

`\parraycell` array cell, and the previous ones are each marked up as `\parraycell{`⟨*cell*⟩`}`, where ⟨*cell*⟩ is the cell content. In last lines of Figure 3 we have used them to create the array lines for `\piece` and `\otherwise`. Note that the array cell specifications in `\parrayline` must coincide with the array specification in the main constructor (here `rl` in `\piecewise`).

3 Limitations

In this section we document known limitations. If you want to help alleviate them, please feel free to contact the package author. Some of them are currently discussed in the sTeX TRAC [**sTeX:online**].

1. none reported yet

4 The Implementation

The `presentation` package generates to files: the LaTeX package (all the code between ⟨*package⟩ and ⟨/package⟩) and the LaTeXML bindings (between ⟨*ltxml⟩ and ⟨/ltxml⟩). We keep the corresponding code fragments together, since the documentation applies to both of them and to prevent them from getting out of sync.

4.1 Package Options

We declare some switches which will modify the behavior according to the package options. Generally, an option `xxx` will just set the appropriate switches to true (otherwise they stay false).[6]

EdN:6

```
1 ⟨*package⟩
2 \ProcessOptions
```

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

```
3 \RequirePackage{keyval}[1997/11/10]
```

[3] Note that this is not a problem when we only run `latex` if we assume that `\piece` and `\otherwise` are only used in arguments of `\piecewise`.

[6] EdNote: we have no options at the moment

```
\RequirePackage{amsmath}
\RequirePackage{etoolbox}
```

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

```
\def\pres@default@precedence{0}
\def\pres@infty{1000000}
\def\iprec{\pres@infty}
\def\niprec{-\pres@infty}
\def\pres@initial@precedence{0}
\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}
```

4.2 The System Commands

EdN:7

`\withprec*` `\withprec` will set the current precedence.[7]

```
\newrobustcmd\withpreci[1]{\edef\pres@current@precedence{#1}}
\newrobustcmd\withprecii[1]{\edef\pres@current@precedence{#1}}
\newrobustcmd\withpreciii[1]{\edef\pres@current@precedence{#1}}
```

EdN:8

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

```
\newrobustcmd\PrecSet[1]{\edef\pres@default@precedence{#1}}
```

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

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

4.3 Prefix & Postfix Notations

We first define the keys for the keyval arguments for `\prefix` and `\postfix`.

```
\def\prepost@clearkeys{%
  \def\pres@p@key{\pres@default@precedence}%
  \def\pres@pi@key{\niprec}%
  \def\pres@lbrack{\pres@default@lbrack}%
  \def\pres@rbrack{\pres@default@rbrack}%
}%
\define@key{prepost}{lbrack}{\def\pres@lbrack{#1}}
\define@key{prepost}{rbrack}{\def\pres@lbrack{#1}}
\define@key{prepost}{p}{\def\pres@p@key{#1}}
\define@key{prepost}{pi}{\def\pres@pi@key{#1}}
```

`\prefix` In prefix we always write the brackets.

```
\newrobustcmd\prefix[3][]{% key, fn, arg
```

[7] EdNote: need to implement this in LaTeXML! it is used in power in smglom/smglom/source/arithmetcis.tex. We also need to document it above!

[8] EdNote: need to implement this in LaTeXML! Also document it above! On the other hand it is never used.

```
  \prepost@clearkeys%
  \setkeys{prepost}{#1}%
  {#2}%
  \pres@lbrack{\edef\pres@current@precedence{\pres@pi@key}#3}\pres@rbrack%
}%
```

\postfix

```
\newrobustcmd\postfix[3][]{%key, fn, arg
  \prepost@clearkeys%
  \setkeys{prepost}{#1}%
  \pres@lbrack{\edef\pres@current@precedence{\pres@pi@key}#3}\pres@rbrack{#2}%
}%
```

4.4 Mixfix Operators

We need to enable notation definitions of the operators that have argument- and precedence-aware renderings. To this end, we circumvent LaTeXML's limitations induced by its internal processing stages, by pulling most of the argument rendering functionality to the XSLT which produces the final OMDoc result.

In the LaTeXML bindings, the internal structure of the mixfix operators is generically preserved, via the `symdef_presentation_pmml` subroutine in the Modules package. Nevertheless, in the current module we add the promised syntactic enhancements to each element of the mixfix family. Also, we use the `argument_precedence` subroutine to store the precedences given by the 'pi', 'pii', etc. keys as a temporary `argprec` attribute of the rendering, to be abolished during the final OMDoc generation. This setup is finally utilized by the XSLT stylesheet which combines the operator structure with the preserved precedences to produce the proper form of the argument render elements.

```
\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}{nobrackets}[yes]{%
  \def\pres@p@key{\pres@infty}%
  \def\pres@pi@key{-\pres@infty}%
}%
\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}%
```

```
}%
```

\mixfixi

```
\newrobustcmd\mixfixi[4][]{%key, pre, arg, post
  \clearkeys%
  \setkeys{mi}{#1}%
  \prep@keys@mi%
  \PrecWrite\pres@lbrack%
  #2{\edef\pres@current@precedence{\pres@pi}#3}#4%
  \PrecWrite\pres@rbrack%
}%
```

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

```
\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}%
    \def\@tmpop{#2}%
  }%
}% update the function
```

\mixfixa

```
\newrobustcmd\mixfixa[5][]{%key, pre, arg, post, assocop
  \clearkeys%
  \setkeys{mi}{#1}%
  \prep@keys@mi%
  \PrecWrite\pres@lbrack{#2}{\@assoc\pres@pi{#5}{#3}}{#4}\PrecWrite\pres@rbrack%
}%
```

EdN:9 \mixfixA A variant of \mixfixa that puts the arguments into an array.[9]

```
\newrobustcmd\mixfixA[5][]{%key, pre, arg, post, assocop
  \clearkeys%
  \setkeys{mi}{#1}%
  \prep@keys@mi%
  \renewrobustcmd\do[1]{\@assoc\pres@pi{#5}{##1}{#5}\tabularnewline}%
  \PrecWrite\pres@lbrack% write bracket if necessary
  #2{\begin{array}{l}\docsvlist{#3}\end{array}}%
  #4\PrecWrite\pres@rbrack%
}%

\define@key{mii}{nobrackets}[yes]{\def\pres@p@key{\pres@infty}%
\def\pres@pi@key{-\pres@infty}\def\pres@pii@key{-\pres@infty}}
\define@key{mii}{lbrack}{\def\pres@lbrack@key{#1}}
\define@key{mii}{rbrack}{\def\pres@lbrack@key{#1}}
```

[9] EdNote: MK: this is very experimental now, if this works, we need to document this above and extend this to the other mixfix declarations. Also we could use a key for the array format argument.

```
\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}%
}%
```

\mixfixii

```
\newrobustcmd\mixfixii[6][]%key, pre, arg1, mid, arg2, post
{\clearkeys\setkeys{mii}{#1}\prep@keys@mii%
\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}
```

\mixfixia

```
\newrobustcmd\mixfixia[7][]%key, pre, arg1, mid, arg2, post, assocop
{\clearkeys\setkeys{mii}{#1}\prep@keys@mii%
\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}
```

EdN:10

\mixfixiA A variant of \mixfixia that puts the arguments into an array.[10]

```
\newrobustcmd\mixfixiA[7][]{%key, pre, arg1, mid, arg2, post, assocop
  \clearkeys\setkeys{mii}{#1}\prep@keys@mii%
  \renewrobustcmd\do[1]{\@assoc\pres@pi{#7}{##1}{#7}\tabularnewline}%
  \PrecWrite\pres@lbrack% write bracket if necessary
  #2{\edef\pres@current@precedence{\pres@pi}#3}%
  #4{\begin{array}{l}\docsvlist{#5}\end{array}}#6%
  \PrecWrite\pres@rbrack%
}%
```

\mixfixai

```
\newrobustcmd\mixfixai[7][]{%key, pre, arg1, mid, arg2, post, assocop
  \clearkeys\setkeys{mii}{#1}\prep@keys@mii%
  \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%
}%

\define@key{miii}{nobrackets}[yes]{\def\pres@p@key{\pres@infty}%
\def\pres@pi@key{-\pres@infty}
\def\pres@pii@key{-\pres@infty}
\def\pres@pii@key{-\pres@infty}}
```

[10] EdNote: MK: this is very experimental now, if this works, we need to document this above and extend this to the other mixfix declarations. Also we could use a key for the array format argument.

```
\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@piii@key}}}
```

\mixfixiii

```
\newrobustcmd\mixfixiii[8][]{%key, pre, arg1, mid1, arg2, mid2, arg3, post
  \clearkeys\setkeys{miii}{#1}\prep@keys@miii%
  \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%
}%
```

\mixfixaii

```
\newrobustcmd\mixfixaii[9][]{%key, pre, arg1, mid1, arg2, mid2, arg3, post, sep
  \clearkeys\setkeys{miii}{#1}\prep@keys@miii%
  \PrecWrite\pres@lbrack% write bracket if necessary
  #2{\@assoc\pres@pi{#9}{#3}}%
  #4{\edef\pres@current@precedence{\pres@pii}#5}%
  #6{\edef\pres@current@precedence{\pres@pii}#7}#8%
  \PrecWrite\pres@rbrack%
}%
```

\mixfixiai

```
\newrobustcmd\mixfixiai[9][]{%key, pre, arg1, mid1, arg2, mid2, arg3, post, assocop
  \clearkeys\setkeys{miii}{#1}\prep@keys@miii%
  \PrecWrite\pres@lbrack% write bracket if necessary
  #2{\edef\pres@current@precedence{\pres@pi}#3}%
  #4{\@assoc\pres@pi{#9}{#5}}%
  #6{\edef\pres@current@precedence{\pres@pii}#7}#8%
  \PrecWrite\pres@rbrack%
}%
```

\mixfixiia

```
\newrobustcmd\mixfixiia[9][]{%key, pre, arg1, mid1, arg2, mid2, arg3, post,assocop
  \clearkeys\setkeys{miii}{#1}\prep@keys@miii%
  \PrecWrite\pres@lbrack% write bracket if necessary
  #2{\edef\pres@current@precedence{\pres@pi}#3}%
  #4{\edef\pres@current@precedence{\pres@pii}#5}%
  #6{\@assoc\pres@pi{#9}{#7}}#8%
  \PrecWrite\pres@rbrack%
}%
```

\prefixa In prefix we always write the brackets.

```
\newrobustcmd\prefixa[4][]{%keys, fn, arg, sep
  \prepost@clearkeys\setkeys{prepost}{#1}%
  {#2}\pres@lbrack{\@assoc\pres@pi@key{#4}{#3}}\pres@rbrack%
}%
```

\postfixa

```
\newrobustcmd\postfixa[4][]{%keys, fn, arg, sep
  \prepost@clearkeys\setkeys{prepost}{#1}%
  \pres@lbrack{\@assoc\pres@pi@key{#4}{#3}}\pres@rbrack{#2}%
}%
```

EdN:11 \infix \infix[11] is a simple special case of \mixfixii.

```
\newrobustcmd\infix[4][]{\mixfixii[#1]{}{#3}{#2}{#4}{}}
```

\assoc

```
\newrobustcmd\assoc[3][]{\mixfixa[#1]{}{#3}{}{#2}}
```

4.5 General Elision

EdN:12 [12]

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

```
\def\setegroup#1#2{\expandafter\def\csname #1@egroup\endcsname{#2}}
```

\elide 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.[13] We test the implementation with Figure 2.

EdN:13

```
\def\elide#1#2#3{%
  \@ifundefined{#1@egroup}%
  {\def\@elevel{0}
   \PackageError{presentation}{undefined egroup #1, assuming value 0}%
   {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}{0}.}%
  }%
  {\edef\@elevel{\csname #1@egroup\endcsname}}%
  \ifnum\@elevel>#2\else{#3}\fi%
}%
```

\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.

[11] EdNote: need infixl as well, use counters for precedences here.
[12] EdNote: all of these still need to be tested and implemented in LaTeXML.
[13] EdNote: do we need to turn this around as well?

par	typ	result	expected
0	0	$\mathbf{I}^{\alpha}{}_{\alpha\to\alpha} := \lambda X_{\alpha}.X$	$\mathbf{I} := \lambda X.X$
600	600	$\mathbf{I} := \lambda X.X$	$\mathbf{I}^{\alpha} := \lambda X_{\alpha}.X$
600	1000	$\mathbf{I} := \lambda X.X$	$\mathbf{I}^{\alpha}_{\alpha\to\alpha} := \lambda X_{\alpha}.X$

Figure 2: Testing Elision with the example in Figure 2

```
\def\provideEdefault#1#2{%
  %\expandafter\def\csname#1@edefault\endcsname{#2}%
  \csdef{#1@edefault}{#2}%
  \@ifundefined{this@module}{}%
  {\expandafter\g@addto@macro\this@module{\expandafter\def\csname#1@edefault\endcsname{#2}}}%
}%
```

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

```
\def\setEdefault#1#2{\expandafter\def\csname #1@edfault\endcsname{#2}}
```

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

```
\def\fromEcontext#1{\csname #1@edefault\endcsname}
```

4.6 Other Layout Primitives

The `\parray`, `\parrayline` and `\parraycell` macros are simple refactorings of the `array` functionality on the LaTeX side.

`\parray`

```
\newrobustcmd\parray[2]{\begin{array}{#1}#2\end{array}}
```

`\parrayline`

```
\newrobustcmd\parrayline[2]{#1#2\\}
```

`\prmatrix`

```
\newrobustcmd\prmatrix[1]{\begin{matrix}#1\end{matrix}}
```

`\pmrow`

```
\def\pmrow#1{\expandafter\@gobble\x@mrow#1\endx@mrow,}
\def\x@mrow#1,{&#1\x@mrow}
\def\endx@mrow#1{\\}
\def\pmrowh#1{\expandafter\@gobble\x@mrowh#1\endx@mrowh,}
\def\x@mrowh#1,{&#1\x@mrowh}
\def\endx@mrowh#1{\\\hline}
```

4.7 Deprecated Functionality

These macros may go away at any time.

\parraylineh

```
\newrobustcmd\parraylineh[2]{#1#2\\\hline}
```

\parraycell

```
\newrobustcmd\parraycell[1]{#1&}
⟨/package⟩
```