Logical markup for mathematical lists

Will Robertson

2007/09/19 v0.5

Contents

1 Introduction

One of LaTeX's lauded features is its separation of form and content. When writing a document, they say, you can just focus on the words. Well, that may be well and good for prose, but mathematicians have frequently been left out in the cold. A funny situation, considering LaTeX's main audience.

Recently, the cool package appeared, marking the first large-scale format-independent method of writing maths in LaTeX. Need to change the brackets surrounding the arguments of `\sin` and `\cos`, or switch from $\tan^{-1}$ to arctan throughout an entire document? No worries.

This sort of initiative will allow much easier transfer of mathematics from such software packages as Mathematica and Matlab, if it becomes popular enough.

This package, mlist, provides a method to write various sorts of mathematical lists without having to worry about formatting. When I say 'lists', I mean things like $\mathbf{A} = (A_1, A_2, A_3)$; to me as a mechanical engineer, vectors and matrices; but the package is more general than that, I hope.

You can use mlist straight away with

```
\usepackage{mlist}
```

and

```
$\vect{A}=\vect{A}{1,2,3}$
```

to get the example in the previous paragraph, but *please* keep reading to learn its more useful features.

2 Basic Functionality

2.1 Prototypical example: vector notation

`\newvect` This example demonstrates why I wrote this package. The control sequence `\dist` is defined as a vector with any number of elements. The vector and its elements can be subsequently referred to without hard-coding any of the mathematical symbols or brackets used.

$\mathbf{D}$ D_ϕ (D_r, D_ϕ, D_θ)

```
\newvect\dist{D}[elem={r,\phi,\theta}]
\[ \dist \qquad \dist{2}
        \qquad \dist{1,2,3} \]
```

2.2 Matrix notation

So the basic idea of this package is to separate the formatting and content of mathematical objects. The above example showed vector notation, and there are two more 'data structures' that are defined 'out of the box'.

\newmatr The second example is for matrices, which one might consider quite similar to vectors; but by default the formatting internally uses amsmath's {bmatrix} environment. Note that it's a very good idea to give your maths objects actual *names* rather than shorthands like \M—who knows if you'll still be calling it 'M' in ten years time?

$\mathbf{M}$ $m + M$ $\begin{bmatrix} m & 0 \\ 0 & m+M \end{bmatrix}$

```
\newmatr\massm{M}[elem={m,0;0,m+M}]
\[ \massm \qquad \massm{4}
          \qquad \massm{1,2;3,4} \]
```

Again, it is possible then to change the brackets used for matrices throughout an entire document (or collection of documents) without changing the fundamental markup of the mathematics itself.

2.3 Function and 'set' notation

\newfunc Finally, here's something a little different. Vector and matrices are both quite similar. But it takes only a slight stretch to extend the syntax of this package to things completely different—such as functions:

$T\,(x_0; x, t)$ $T\,(x)$ $T\,(x, t)$

```
\newfunc\traj{T}[elem={x_0;x,t},index={1;2,3}]
\[ \traj \qquad \traj{2} \qquad \traj{2,3} \]
```

For example, Mathematica users may wish to typeset their arguments with square brackets in their own documents but not for published papers. To hammer home the point: this can be now done for an entire document with a simple switch.

\newmset Here's an example to further demonstrate the flexibility of the package.

$\mathbb{T}$ $\mathbb{T}^2$ $\mathbb{T}^{2\times 3}$

```
\newmset\setT{T}
\[ \setT \qquad \setT{2} \qquad \setT{2,3} \]
```

The \setR, \setC, \setN, and \setZ 'msets' are defined by default for real, complex, natural, and integer numbers.

$\boldsymbol{x} \in \mathbb{R}^{2\times 2}$

```
\[ \bm{x}\in\setR{2,2} \]
```

2.4 Non-specific commands

Sometimes you don't always want to define new macros for one-off maths expressions. Accompanying the \newvect, \newmatr, \newfunc, and \newmset

macros are `\vect`, `\matr`, `\func`, and `\mset`, which can be used for single cases:

$$(V_a, V_b, V_c) \quad \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad f\left(x_0; x, t\right) \quad \mathbb{D}^{m \times n}$$

```
\[ \vect{V}{a,b,c}   \quad
   \matr{M}{a,b;c,d} \quad
   \func{f}{x_0;x,t} \quad
   \mset{D}{m,n}     \]
```

This usage is generally not recommended except in isolated cases.

2.5 An example of some actual maths

Here's a practical example using `\matr` that I think demonstrates well the advantages of using this package:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \dot{x}_1 \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -K/M & -C/M \end{bmatrix}$$

$$\frac{\partial}{\partial t}\begin{bmatrix} x_1 \\ \dot{x}_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -K/M & -C/M \end{bmatrix}\begin{bmatrix} x_1 \\ \dot{x}_1 \end{bmatrix}$$

```
\newmatr\state{x}
  [elem={x_1;\dot x_1},index={1;2}]
\newmatr\dyn{A}
  [elem={0,1;-K/M,-C/M},
   index={1,2;3,4}]
\[ \dot{\state{}} = \dyn{}\state{} \]
\[ \state{} = \state \]
\[ \dyn{} = \dyn \]
\[ \frac\partial{\partial t}
    \state = \dyn \state \]
```

There's a lot more semantic meaning in that mathematical TEX source than if it were hard-coded without this package.

2.6 Elements and indexing

For context for the following explanations, here's our first example again:

$$\mathbf{D} \qquad D_\phi \qquad \left(D_r, D_\phi, D_\theta\right)$$

```
\newvect\dist{D}[elem={r,\phi,\theta}]
\[ \dist \qquad \dist{2} \qquad \dist{1,2,3} \]
```

Each command created like `\dist` takes zero or one arguments, as shown above. Unlike regular LATEX commands with optional arguments, these still use curly braces, and an empty argument is equivalent (by default) to omitting the argument altogether:

`\dist{}` → $\mathbf{D}$ ≡ `\dist` → $\mathbf{D}$.

While numerical indexing is used to extract `elem` items from the list, *any* argument can be used instead that is passed through directly:

`\dist{x^2}` → D_{x^2} .

Table 1: How two-dimension lists elements are indexed. The first row shows the input list via `elem={}`. The second and third rows respectively show the direct index and implicit index required to reference the elements of the list.

List	{ a , b , c ; d , e ; f }
Direct index	{ 1 , 2 , 3 ; 4 , 5 ; 6 }
Implicit index	{ @1 , @2 , @3 ; @1 , @2 ; @1 }

As we saw for the matrix examples, the input may actually be 'two-dimensional'; as well as being comma-separated, elements (and sets of elements) may also be separated by semi-colons:

`\dist{x;y;z}` → $\left(D_x;D_y;D_z\right)$.

Input elements can also be specified with semi-colon separators to match up with two-dimensional element indexing. There are two ways to reference these elements: with a linear index from one to the number of elements in `elem`; or with 'implicit indexing' that starts from one for every semi-colon list. Implicit indices are prefix with the `@` symbol.[1] An example is useful to show how this works:

$\mathbf{V} = \left(V_a, V_b, V_c; V_x, V_y, V_z\right)$

$\mathbf{V} = \left(V_x, V_y, V_z\right)$

$\mathbf{V} = \left(V_a, V_b, V_c; V_x, V_y, V_z\right)$

```
\newvect\V{V}[elem={a,b,c;x,y,z}]
\[ \V=\V{1,2,3;4,5,6} \]
\[ \V=\V{4,5,6} \]
\[ \V=\V{1,2,3;@1,@2,@3} \]
```

See table 1 for a more detailed example of the indexing system.

You can also set the default element list when defining a vector. In this case, an empty list argument is not equivalent to omitting one:

$\left(L_i, L_j, L_k\right)$ vs. $\mathbf{L}$ vs. $\left(L_i, L_j\right)$

```
\newvect\LL{L}[elem={i,j,k},index={1,2,3}]
$\LL$ \quad vs.\quad $\LL{}$
      \quad vs.\quad $\LL{1,2} $
```

This was shown originally in the function example on page 3.

2.7 Shorthand indices and empty elements

Shorthands are defined to allow things that *aren't* elements into the list. In this example, the semi-colon separation is shown with the `\dots` shorthand '`:`' and the `\cdot` shorthand '`.`'.

[1] The implicit indexing syntax will work *even if* `\makeatletter` is in effect.

$\mathbf{M} = (M_1, M_2, \ldots, M_5; M_a, M_b, \cdot, M_d)$

```
\newvect\mm{M}
\[ \mm=\mm{1,2,:,5;a,b,.,d} \]
```

More shorthands will be defined in the future via an extensible mechanism that does not yet exist.

Empty arguments are ignored (where 'empty' ≡ empty or whitespace).

$\mathbf{N} = (N_1, N_3, N_5; N_2)$

```
\newvect\nset{N}
\[ \nset=\nset{1,,,3, ,5;;,2,;} \]
```

2.8 \dots

Finally, some macros are defined in order to be able to typeset unknown ranges $(1, 2, \ldots, N)$ without hard-coding the symbol of the maximum element. Similarly, it is also useful to denote a 'mid-range' element somewhere inside the ellipses. Note the equivalence in defining the mid and last elements and indexing them.

$\mathbf{W} = (W_i, W_j, \ldots, W_{w-1}, W_w)$

$\mathbf{Y} = (Y_i, Y_j, \ldots, Y_p, \ldots, Y_y)$

```
\newvect\W{W}[elem={i,j,k,:,w}]
\newvect\Y{Y}[elem={i,j,k,:,p,:,y}]
\[ \W=\W{1,2,:,\LAST-1,\LAST} \]
\[ \Y=\Y{1,2,:,\MID,:,\LAST} \]
```

The \MID and \LAST control sequences are 'implicit' for two dimensional lists:

$(X_i, \ldots, X_m, \ldots, X_w; X_{ii}, \ldots, X_{mm}, \ldots, X_{ww})$

$(X_i, X_j, X_k; X_{ii}, X_{jj}, X_{kk})$

```
\newvect\X{X}[%
  elem={i,j,k,:,m,:,w;ii,jj,kk,:,mm,:,ww}]
\[ \X{1,:,\MID,:,\LAST;@1,:,\MID,:,\LAST} \]
\[ \X{1,2,3;4,5,6} \]
```

Finally, note well above that any {:,X} pairs for mid/last element definitions do *not* add to the direct index numbering.

3 Generalisation

\newmlist
\renewmlist

The \newvect command that has been often shown previously is an example of a macro created with \newmlist, which can be considered something like an *instance generator* for the types of lists we're dealing with. Use \renewmlist if the command is already in use.

\newmlist\<list>[⟨*list options*⟩] creates macros \<list> and \new<list> (and \renew<list>) that are analogous to \vect and \newvect seen in section 2.

⟨*list options*⟩ can contain any of those seen so far in previous `\newvect` macros, plus more to be introduced soon.

In the examples to follow, `\vect` and `\newvect` are often still used to demonstrate various options, overriding the defaults.

3.1 Head and element formatting

The `headcmd` and `elemcmd` options are used to alter the formatting of the vector symbol and its elements. They are passed *macros* that take, respectively, one and two arguments that define the formatting.

For `\vect`, `headcmd=\mathbf` and `elemcmd=\mlistsub`, where

`\mlistsub{#1}{#2}` → `#1_{#2}`.

`\mlistsup`, the analogous command for creating superscripts, is defined by the package in case you need it.

In this example, the vector symbol is formatted with an arrow accent, and the elements are exactly as specified in `elem`.

$\vec{A}$ (a,b,c,d) c

```
\newvect\A{A}[%
  headcmd=\vec,
  elemcmd=\mlistelem,
  elem={a,b,c,d}]

\[\A \qquad \A{1,2,3,4} \qquad \A{3} \]
```

The `\mlistelem` command (with its friend, `\mlisthead`) is defined as follows:

`\mlistelem{#1}{#2}` → `#2`

`\mlisthead{#1}` → `#1`

In section 2.6, we saw that list indexing doesn't *have* to be numerical. In this example, the use of `\@alph` allows alphabetic subscripts but non-numeric indices will produce errors:

(A_a, A_b, A_c, A_d) A_c

```
\makeatletter
\newcommand\subalph[2]{#1_{\@alph{#2}}}
\makeatother
\newvect\A{A}[%
  elemcmd=\subalph]

\[ \A{1,2,3,4} \qquad \A{3} \]
```

3.2 Delimiter formatting

The `wrap` and `wrapone` options are used to change the way elements are displayed together. Each take two arguments to define the opening and closing

material.

By default, if there is more than one element, it is surrounded by square brackets: `wrap=[]`. A single element is typeset naked: `wrapone={}{}`. If `wrapone` is called with no '=' argument, it takes the same value as `wrap`.

In this example, the brackets around the sets are changed to parentheses, including single elements. The behaviour for a single element is then changed to use angle brackets:

$\mathbf{Q} + Q_1 + (Q_a, Q_b)$
$\mathbf{Q} + (Q_1) + (Q_a, Q_b)$
$\mathbf{Q} + \langle Q_1 \rangle + (Q_a, Q_b)$

```
\newvect\Q{Q}
$\Q + \Q{1} + \Q{a,b}$

\newvect\QQ{Q}[wrap=(),wrapone]
$\QQ + \QQ{1} + \QQ{a,b}$

\newvect\QQQ{Q}[wrapone={\left<}{\right>}]
$\QQQ + \QQQ{1} + \QQQ{a,b}$
```

Rather than defining the open/close material around a set of elements, it can often be necessary to define a macro that defines the formatting of the set. The `wrapcmd` and `wraponecmd` options are used for this purpose, and take as argument a single macro that accepts two arguments: the list head and the list elements, respectively.

$\mathbf{R} + R_1 + (R_a, R_b)$
$\mathbf{R} + \langle R_1 \rangle_{\mathbf{R}} + \langle R_a, R_b \rangle_{\mathbf{R}}$
$\mathbf{R} + \{R_1\}^{\mathbf{R}} + \langle R_a, R_b \rangle_{\mathbf{R}}$

```
\newvect\R{R}
$\R + \R{1} + \R{a,b}$

\newcommand\mywrap[2]{\langle{\color{red}#2}\rangle_{#1}}
\newvect\RR{R}[wrapcmd=\mywrap,wraponecmd]
$\RR + \RR{1} + \RR{a,b}$

\newcommand\mywrapone[2]{\{{\color{green}#2}\}^{#1}}
\newvect\RRR{R}[
  wrapcmd=\mywrap,
  wraponecmd=\mywrapone]
$\RRR + \RRR{1} + \RRR{a,b}$
```

mlist provides some example commands for this purpose:

```
\mlistnowrap     → #2
\mlistparen      → \left (#2\right )
\mlistbrack      → \left [#2\right ]
\mlistbrace      → \left \{#2\right \}
\mlistangle      → \left <#2\right >
\mlistheadparen  → #1\left (#2\right )
\mlistheadbrack  → #1\left [#2\right ]
\mlistheadbrace  → #1\left \{#2\right \}
\mlistheadangle  → #1\left <#2\right >
```

3.3 Separator formatting

The `sep` and `sepsep` options take one argument that is inserted between items in comma and semi-colon lists, respectively. For example, in the `\matr` list these are defined with `[sep=&, sepsep=\\]`.

Here's another example:

$$S_a + S_b + S_c + S_d + \cdots + S_p + S_q + S_r + S_s + \cdots + S_x + S_y + S_z$$

```
\newvect\mysum{S}[
  sep=+,
  wrap={}{},
  dots=\cdots]
$\mysum{a,b,c,d,:,p,q,r,s,:,x,y,z}$
```

3.4 Global options definition

`\mlistsetup` If options aren't specified in `\newmlist` they are inherited from the global defaults, which may be adjusted with `\mlistsetup{`⟨*mlist options*⟩`}`. The defaults are shown in section 5 on the following page.

3.5 Redefining `\vect`, `\matr`, `\func`, `\mset`

This package makes little claim for being imminently usable for most people *out of the box*. I figure there's just too much variety; people need to define their own 'mlists' with `\newmlist`.

If changes to the mlists provided by default with this package are required (as they will be if any aspect of their formatting needs to be adjusted), simply create a local `mlist.cfg` file with different defintions or even just copy them from section 5 on the next page in a `\renewmlist`.

4 TODO

- changing features of (or adding features to, rather) 'newvect' constructions, rather than overwriting them.
- generalise escaping strings like :→\dots, etc.
- accents and appended/prepended material migrating inside the head (and/or each element) argument.
- optional arguments for \MID and \LAST (for explicit indexing).
- coercing one type of list into another (and retrieving non-wrapped lists as a special case)
- use ltx3!

5 The default definitions

This is the code that appears in the default configuration file mlist.cfg to set up the default options and mlists. Edit a copy of mlist.cfg in a local location (in a local texmf tree or on a per-document basis) to change these definitions and to create your own mlists.

5.1 Global options

These are inherited by mlists that do not explicitly define their own respective options. Geared towards \vect, essentially.

```
\mlistsetup{%
  sep={,},
  sepsep={;},
  wrapcmd=\mlistparen,
  wrapone={}{},
  dots=\dots,
  dot={\,\cdot\,},
  elemcmd=\mlistelem,
  headcmd=\mathbf,
  index={},
}
```

5.2 Provided mlists

Redefine these with \renewmlist to effect your own formatting of these mlists. Note that these assume certain global defaults (see above), so some aspects of their formatting can be changed with \mlistsetup.

`\vect` By default `\vect{V}=\vect{V}{a,b,c}` → $\mathbf{V} = (V_a, V_b, V_c)$.

```
\newmlist\vect[
  wrapcmd=\mlistparen,
  wraponecmd=\mlistnowrap,
  elemcmd=\mlistsub,
  headcmd=\mathbf,
]
```

`\matr` By default `\matr{M}=\matr{M}{a,b;c,d}` → $\mathbf{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$.

```
\newmlist\matr[
  sep=&,
  sepsep=\\,
  wrap={\begin{bmatrix}}
       {\end{bmatrix}},
]
```

`\func` By default, `\func{f}=\func{f}{x,y,z}` → $f = f\,(x, y, z)$.

```
\newmlist\func[
  headcmd=\mlisthead,
  wrapcmd=\mlistheadparen,
  wraponecmd,
]
```

`\mset` By default, `\mset{N}{2,3}` → $\mathbb{N}^{2\times3}$.

```
\newmlist\mset[
  headcmd=\mathbb,
  sep=\times,
  wrapcmd=\mlistsup,
  wraponecmd,
]
```

Sets for real, complex, natural, and integer numbers, respectively:

```
\newmset\setR{R}
\newmset\setC{C}
\newmset\setN{N}
\newmset\setZ{Z}
```

6 Prior art

With CTAN getting so big these days, it's quite necessary to undertake extensive literature reviews before writing your own package. I've been burned before, spending a couple of days playing with ideas and then realising that someone's already done what I wanted. So this time I looked before I jumped.

\bvec	\buvec	\svec	\suvec	\uvec	\uuvec	\irvec
$\mathbf{a}$	$\hat{\mathbf{a}}$	a	$\hat{\mathsf{a}}$	$\underline{a}$	$\underline{\hat{a}}$	$a_1, \ldots, a_n$

Table 2: Commands defined by the vector package. \uvec and \uuvec can be configured to produce an under-tilde instead. \irvec takes an optional argument for 'n' and can be configured (globally) to begin from a different index.

easyvector The most similar package to this one (that I could find) is easyvector.[2] Superficially, there are a number of similarities, but it didn't quite do what I wanted. Here's an example demonstrating the creation of new vector macros:

$\mathbf{Z} = (Z^k_{i,j})$ $\quad Z_{1,2,3} \neq \mathbf{Z}_{1,2,3}$

```
\newvector[Z,\mathbf{Z}]{X}
\[ \X = (\X[i,j;k]) \qquad
   \X[1,2,3] \neq \X![1,2,3] \]
```

Note that each macro refers only to a single vector element. This is a package to simplify input (and abstract formatting) of vectors with complex notation. easyvector also allows you to customise the form of the vectors it produces, but this is rather inflexible without a good deal of work.

$\mathbf{b}$ $\mathrm{b}^3_{1_2}$ $\mathrm{b}^1_{3_2}$

```
\def\myindex[#1,#2,#3]{_{#1_{#2}}^{#3}}
\newcustomvector[\mathtt{b},\mathbf{b}]{bb}\myindex
\[ \bb \qquad \bb[1,2,3] \qquad \bb[3,2,1] \]
```

Note also that the main symbol ('b', here) is not available even in this case, so prepended sub-/superscripts are not possible.

Has the interesting option to reference matrix row/column sub-vectors: (careful to ensure \makeatother manually)

$\mathbf{W} = (\mathbf{W}_{\bullet,j})$ $\quad \mathbf{W}^{\bullet}_{a,b}$

```
\newvector(W)[wvec]
\[ \wvec = (\wvec[@,j]) \qquad \wvec[a,b;@] \]
```

I like this idea of 'shorthand' symbols.

vector The package vector[3] provides a few commands for setting vector and matrix symbols, shown in table 2. Just a few basic macros to simplify input, but not really to separate form and content.

vector provided the inspiration to add \dots ideas to this package. It also highlights that something specific should be done with accents in general and provide a content macro for unit vectors.

[2] http://tug.ctan.org/cgi-bin/ctanPackageInformation.py?id=easyvector
[3] http://tug.ctan.org/cgi-bin/ctanPackageInformation.py?id=vector

hhtensor The hhtensor package[4] provides a few content-based macros with global options to specify their appearance. `\vec`, `\matr` and `\tens` are provided for vectors, matrices, and tensors, with mathematical symbols `\dcdot` to denote double scalar products, and `\trans` for printing an upright superscript 'T' to denote the transpose operator.

This package was the inspiration for pre-defined macros for specific meanings. While I approve of defining the transpose symbol, it's outside the scope of mlist.

Other tensor packages There are three main packages for typesetting tensors. These start to stray from the interest of mlist. The relevant packages are tensind,[5] tensor,[6] and mattens.[7] Each focuses on typesetting various forms of tensor notation, which differ from what this package is trying to do. Integration between this package and those three above are possible, but has not yet been investigated.

[4] http://tug.ctan.org/cgi-bin/ctanPackageInformation.py?id=hhtensor
[5] http://tug.ctan.org/cgi-bin/ctanPackageInformation.py?id=tensind
[6] http://tug.ctan.org/cgi-bin/ctanPackageInformation.py?id=tensor
[7] http://tug.ctan.org/cgi-bin/ctanPackageInformation.py?id=mattens

File I
mlist implementation

7 Setup code

This is the package.

```
\ProvidesPackage{mlist}
  [2007/09/19 v0.5 Typesetting maths lists]
```

Change History

v0.1
General: Code tidy up; first decent version. 14
v0.2
General: More documentation; list indexing not finalised. 14
v0.3
General: List indexing decided; empty arguments ignored. 14
Tidied up the \MID/\LAST element stuff a bit. 18
v0.4
\@mlist: Eliminated \global no longer required since I separated the list creation and typesetting. 20
Simplified 'single index' code. 20
\@optarg: Implemented to simplify optional arg processing. 16
General: Tidied things up a little bit. 14
v0.5
\@mlist: Added \mlist@headcmd command to the \mlist@symbolinside \mlist@wrap (fixed bug with \mset) 20
\mset: Added. 11

7.1 User shorthands

For headcmd:

```
\let\mlisthead\@firstofone
```

For elemcmd:

```
\let\mlistelem\@secondoftwo
\newcommand\mlistsub  [2]{#1_{#2}}
\newcommand\mlistsup  [2]{#1^{#2}}
```

For wrapcmd:

```
\let\mlistnowrap\@secondoftwo
\newcommand\mlistparen[2]{\left(#2\right)}
\newcommand\mlistbrack[2]{\left[#2\right]}
```

```
\newcommand\mlistbrace[2]{\left\{#2\right\}}
\newcommand\mlistangle[2]{\left<#2\right>}
\newcommand\mlistheadparen[2]{#1\left(#2\right)}
\newcommand\mlistheadbrack[2]{#1\left[#2\right]}
\newcommand\mlistheadbrace[2]{#1\left\{#2\right\}}
\newcommand\mlistheadangle[2]{#1\left<#2\right>}
```

7.2 Packages

```
\RequirePackage{xkeyval,ifmtarg}
```

7.3 Code we need

Conditionals and counters and things:

```
\newif\if@mlist@notfirst@
\newif\if@mlist@implicit@
\newcount\mlist@elem@N
\def\@gobblenil#1\@nil{}
```

Some specific things:

```
\newif\if@mlist@warn
```

Semi-colon delimited iteration (adapted from ltx2e).

```
\long\def\@sfor#1:=#2\do#3{%
  \expandafter\def\expandafter\@sfortmp\expandafter{#2}%
  \ifx\@sfortmp\@empty\else
    \expandafter\@sforloop#2;\@nil;\@nil\@@#1{#3}%
  \fi}
\long\def\@sforloop#1;#2;#3\@@#4#5{%
  \def#4{#1}%
  \ifx #4\@nnil \else
    #5%
    \def#4{#2}%
    \ifx #4\@nnil \else
      #5%
      \@siforloop #3\@@#4{#5}%
    \fi
  \fi}
\long\def\@siforloop#1;#2\@@#3#4{%
  \def#3{#1}%
  \ifx #3\@nnil
    \expandafter\@fornoop
  \else
    #4\relax\expandafter\@siforloop
  \fi
  #2\@@#3{#4}}
```

ltx3-inspired syntax. `\def@c` still needs arguments to be supplied to it.

```
\providecommand\let@cc[2]{%
  \expandafter\let\csname#1\expandafter\endcsname\csname#2\endcsname}
\providecommand\def@c[1]{%
  \expandafter\def\csname#1\endcsname}
\providecommand\def@co[2]{%
  \def@c{#1\expandafter}\expandafter{#2}}
```

`\@optarg` Macro to simplify optional argument parsing.

```
\newcommand\@optarg[1]{\@ifnextchar[{#1}{#1[]}}%]
```

Shorthand to test for optional brace arguments:

```
\newcommand\@ifnextbrace{\expandafter\@ifnextchar\bgroup}
```

Shorthand string definitions for ifx tests:

```
\def\mlist@colon{:}
\def\mlist@period{.}
```

This is for checking for `@` with an 'other' catcode:

```
\makeatother
\expandafter\def\csname mlist@ampersat\endcsname{@}
\makeatletter
```

('Ampersat' is a name for `@` I find amusing. That symbol doesn't seem to have a definitive official name.)

8 keyval options

`\mlistsetup` `#1` : keyval options
User command to set global defaults for mlists.

```
\newcommand\mlistsetup[1]{\setkeys[mlist]{sym}{#1}}
```

xkeyval makes it easy for us to define a whole slew of options that simply save their argument to a macro.

```
\define@cmdkeys[mlist]{sym}[mlist@]{%
  symbol,index,sep,sepsep,elem,
  elemcmd,headcmd,dot,dots}
```

8.1 Wrapping

The wrapcmd options also just save their argument, but wraponecmd takes wrapcmd as a default:

```
\define@key[mlist]{sym}{wrapcmd}{\let\mlist@wrap#1}
\define@key[mlist]{sym}{wraponecmd}[\mlist@wrap]{\let\mlist@wrapone#1}
```

The wrap and wrapone options need a bit more logic in them:

```
\define@key[mlist]{sym}{wrap}{%
  \def\mlist@wrap##1##2{\@firstoftwo#1##2\@secondoftwo#1}}
\define@key[mlist]{sym}{wrapone}[]{%
  \ifx\relax#1\relax
    \let\mlist@wrapone\mlist@wrap
  \else
    \def\mlist@wrapone##1##2{\@firstoftwo#1##2\@secondoftwo#1}%
  \fi}
```

8.2 Element definition

Takes as input a two dimension list with comma-separated elements and semicolon-separated lists of elements:

`#1` → `{1,2,3;4,5;6}` .

Elements could be numerical or arbitrary TEX code.

```
\define@key[mlist]{sym}{elem}{%
```

It would be easier for all involved if I used specific counters for the following. For now, they're generic and harder to comprehend in six months:
`\@tempcnta` Number of semicolon list.
`\@tempcntb` Element number of this comma list.

```
  \@tempcnta\z@
  \@tempcntb\z@
  \mlist@elem@N\z@
```

Iterate over every semicolon list. Set `\if@tempswa` true only after every `:` element, for each semicolon list. Maybe we should use a counter for this, instead?

```
  \@sfor\@jj:=#1\do{%
    \advance\@tempcnta\@ne
    \@tempcntb\z@
    \@tempswafalse
```

Iterate over every comma list:

```
    \@for\@ii:=\@jj\do{%
```

If the element is `:`, set the switch for the next comma-iteration.

```
      \ifx\@ii\mlist@colon
        \@tempswatrue
      \else
```

If the previous element was `:`, save the 'last' element: (TODO: add error check for too many `:`). If we've already defined the 'last' element and run into `:` again, that means we actually wanted 'mid' so make the redefinition. This should only happen once anyway so it can occur every time `\if@tempswa` is true.

```
        \if@tempswa
          \let@cc{mlist@\the\@tempcnta @mid}{mlist@\the\@tempcnta @last}%
```

```
            \def\@tempa{\def@c{mlist@\the\@tempcnta @last}}%
            \expandafter\@tempa\expandafter{\@ii}%
          \else
```

Otherwise, bump up the counters and define the elements. There are two definitions we used for the element indexing. As an example, if we're up to index $(3,1)$ of a list like $\{a,b,c;d,e;f\}$ then we define

\mlist@3@1 $\rightarrow$ elem f , and
\mlist@@6 $\rightarrow$ elem f .

(Index 3,1 is the sixth in the list.)

```
            \advance\@tempcntb\@ne
            \advance\mlist@elem@N\@ne
            \def@co{mlist@\the\@tempcnta @\the\@tempcntb}{\@ii}%
            \def@co{mlist@@\the\mlist@elem@N}{\@ii}%
          \fi
        \fi}%
```

If swa is true, we are in an element after a `:`, so turn it off. TODO: I guess no more regular elements can turn up anyway so this is probably overkill!

```
    \if@tempswa\@tempswafalse\fi}}
```

8.3 Shorthand definitions

Currently defunct:

```
\define@key[mlist]{sym}{shorthand}{%
  \mlist@def@shorthand#1%
  \def@c{mlist@@\@tempa\expandafter}\expandafter{\@tempb}}
\def\mlist@def@shorthand#1#{%
  \def\@tempa{\string#1}
  \def\@tempb}
```

9 List indexing

How do I want indexing to work? Originally, the input was a one dimensional list with the output directly following the indexing:

```
A = {a1,a2,a3,a4}
A{1,2;3,4} == {a1,a2;a3,a4}
```

I tried a couple of other things, but went back to this idea plus the implicit indexing.

`\@mlist` `#1 :` keyval options

This is the \setkeys wrapper, which sets macros for us in various ways depending on how it has been called and then moves on to do the actual list indexing, extraction, and typesetting.

```
\newcommand\@mlist[1]{%
  \begingroup
  \let\mlist@list@elems\@empty
  \@tempcnta\@ne
  \mlist@count
```

The \setkeys code to extract our mlist is flanked by some scary code to protect ourselves inside things like {array} environments. Thanks Morten.

```
  \iffalse{\fi\ifnum0=‘}\fi
  \setkeys[mlist]{sym}{#1}%
  \ifnum0=‘{\fi\iffalse}\fi
```

These macros only exist inside an mlist index:

```
  \def\MID{\csname mlist@\the\@tempcnta @mid\endcsname}%
  \def\LAST{\csname mlist@\the\@tempcnta @last\endcsname}%
```

If there are no optional arguments, typeset the plain vector symbol.

```
  \ifx\@empty\mlist@index\relax
    \def\mlist@list{\mlist@headcmd{\mlist@symbol}}%
  \else
```

Otherwise, iterate over every semicolon-separated list.

```
    \@tempcnta\z@
    \@tempswafalse
    \@sfor\@jj:=\mlist@index\do{%
      \let\@jj\@jj
      \advance\@tempcnta\@ne
      \mlist@count
      \expandafter\@ifmtarg\expandafter{\@jj}{}% ignore if empty
        {\if@tempswa\mlist@add\mlist@sepsep\fi
         \@tempswatrue}%
```

And (sub-)iterate over every comma-separated list:

```
      \@tempcntb\z@
      \@mlist@notfirst@false
      \@for\@ii:=\@jj\do{%
        \let\@ii\@ii
        \advance\@tempcntb\@ne
```

Now grab the symbol we’re up to. When it’s empty:

```
        \expandafter\@ifmtarg\expandafter{\@ii}%
          {\advance\@tempcntb\m@ne
           \mlist@add{\@gobble}}
```

When it’s not empty:

```
          {\if@mlist@notfirst@\mlist@add\mlist@sep\fi
```

```
            \@mlist@notfirst@true
            \ifx\@ii\mlist@colon
              \mlist@add{\mlist@dots\@gobble}%
            \else
              \ifx\@ii\mlist@period
                \mlist@add{\mlist@dot\@gobble}%
              \else
                \mlist@add{\mlist@elemcmd{\mlist@symbol}}%
              \fi
            \fi}%
```

Parse the index to see if it's an implicit reference of the form `@`n. @mlist@implicit@ is true if the index is implicit: (and we assume `@` will be only used in an implicit-indexing context. TODO: fix this!)

```
          \mlist@parse@implicit
          \if@mlist@implicit@
            \mlist@add@ifcs{%
              mlist@\the\@tempcnta @\expandafter\@gobble\@ii
            }{\@ii}%
          \else
```

Because `\@ii`, `\@jj` can contain any possible index that might be passed through the macro, we use eTeX's `\detokenize` to prevent expansion of any weird argument that might be given.

```
            \mlist@add@ifcs{%
              mlist@@\expandafter\detokenize\expandafter{\@ii}%
            }{\@ii}%
          \fi
        }%
```

That was the end of comma-separated iteration.

```
      }%
```

That was the end of semicolon-separated iteration.

```
    \def\mlist@list{%
      \mlist@wrap{\mlist@headcmd{\mlist@symbol}}
                 {\mlist@list@elems}}%
```

If only have a single element:

```
    \ifnum\@tempcnta=\@ne
      \ifnum\@tempcntb=\@ne
        \let\mlist@wrap\mlist@wrapone
      \fi
    \fi
  \fi
  \mlist@list
  \endgroup}
```

TODO: move `\mlist@list` after the endgroup?

`\mlist@add` Used in the above to build up the `\mlist@list` list.

```
\newcommand\mlist@add[1]{%
  \expandafter\gdef
  \expandafter\mlist@list@elems
  \expandafter{\mlist@list@elems#1}}
```

`\mlist@count` This macro is used to increment the counter used in `\MID`/`\LAST` during the actual typesetting of the indexed elements.

```
\def\mlist@count{%
  \expandafter\mlist@add
  \expandafter{%
  \expandafter\@tempcnta\the\@tempcnta\relax}}
```

`\mlist@add@ifcs` Pretty specific macro to save some repetition. Uses `\mlist@add` to add the first argument as a csname if it exists, otherwise adds the literal second argument.

```
\newcommand\mlist@add@ifcs[2]{%
  \ifcsname#1\endcsname
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi{\expandafter\mlist@add\expandafter{\expandafter{\csname#1\endcsname}}}
     {\expandafter\mlist@add\expandafter{\expandafter{#2}}}}
```

`\mlist@parse@implicit` Parses the index to see if it's an implicit reference: `@`*n*. The idea is to expand out the index, set a conditional if the first char is an `@`, and then gobble up the whole thing. We do this twice for the two common catcodes that `@` might be.

```
\def\mlist@parse@implicit{%
  \@mlist@implicit@false
  \def\@tempb{%
    \expandafter\@ifnextchar\mlist@ampersat
      {\@mlist@implicit@true\@gobblenil}
      {\@mlist@implicit@false\@gobblenil}}%
  \expandafter\@tempb\@ii\@nil
  \unless\if@mlist@implicit@
    \def\@tempb{%
      \@ifnextchar @
        {\@mlist@implicit@true\@gobblenil}
        {\@mlist@implicit@false\@gobblenil}}%
    \expandafter\@tempb\@ii\@nil
  \fi}
```

10 List creation

This is the idea behind the user level macros of the package:

```
\newmlist\vect{all-vect-opts}
\vect{v}{ind} == \@mlist{symbol={v},all-vect-opts,index={ind}}

\newvect\vv{v}[this-v-options]
\vv[ind] == \@mlist{symbol={v},all-vect-opts,this-v-options,index={ind}}
```

\newmlist
\renewmlist

#1 : control sequence
[⟨*keyval options*⟩] Things get pretty hairy because I have to do all the optional argument processing manually. I'd be better off using the suffix package, but the code works for now. LTX3 will make things much easier.

```
\newcommand\newmlist[1]{%
  \@mlist@warntrue
  \@optarg{\@defmlist@opt{#1}}}
\newcommand\renewmlist[1]{%
  \@mlist@warnfalse
  \@optarg{\@defmlist@opt{#1}}}
\def\@defmlist@opt#1[#2]{\@defmlist{#1}{#2}}
```

Processing #1 = thislist-cmd, #2 = all-thislist-opts

```
\newcommand\@defmlist[2]{%
  \if@mlist@warn
    \ifdefined#1
      \PackageError{mlist}{Command \string#1 already defined}
        {\string#1 will be overwritten}%
    \fi
  \fi
```

Plain usage: (e.g., \vect{a}{1,2,3})

```
  \protected\def#1##1{%
    \@ifnextbrace
      {\csname\string#1@opt\endcsname{##1}}
      {\csname\string#1@opt\endcsname{##1}{}}}%
  \def@c{\string#1@opt}##1##2{%
    \csname\string#1\endcsname{##1}{##2}}%
  \def@c{\string#1}##1##2{%
    \@mlist{symbol=##1,#2,index={##2}}}%
```

mlist macro definitions: (e.g., \newvect{A}[...]) Processing ##1 = this-sym-cmd, ##2 = symbol, ##3 = this-sym-options

```
  \def@c{new\expandafter\@gobble\string#1}##1##2{%
    \@mlist@warntrue
    \@optarg{\csname @new\string#1@opt\endcsname{##1}{##2}}}%
  \def@c{renew\expandafter\@gobble\string#1}##1##2{%
    \@mlist@warnfalse
    \@optarg{\csname @new\string#1@opt\endcsname{##1}{##2}}}%
  \def@c{@new\string#1@opt}##1##2[##3]{%
```

```
    \csname @new\string#1\endcsname{##1}{##2}{##3}}%
  \def@c{@new\string#1}##1##2##3{%
    \if@mlist@warn
      \ifdefined##1
        \PackageError{mlist}{Command \string##1 already defined}
          {You cannot overwrite previous definition of \string##1}%
      \fi
    \fi
    \protected\def##1{%
      \@ifnextbrace
        {\csname @new\string##1@opt\endcsname}
        {\@mlist{symbol=##2,#2,##3}}}%
    \def@c{@new\string##1@opt}####1{%
      \@mlist{symbol=##2,#2,##3,index={####1}}}}%
  }
```

MLIST: symbol=S,opt1=one,

MLIST: symbol=S,opt1=one,,index=1,2,3

MLIST: symbol=T,opt1=one,opt2=two

MLIST: symbol=T,opt1=one,opt2=two,index=1,2,3

```
\def\@mlist#1{MLIST: #1\par}
\newmlist\testlist[opt1=one]
\newtestlist\testone{S}
\testone
\testone{1,2,3}
\newtestlist\testtwo{T}[opt2=two]
\testtwo
\testtwo{1,2,3}
```

```
\InputIfFileExists{mlist.cfg}{}
 {\PackageWarning{mlist}{No configuration file (mlist.cfg) found nor loaded}}
```