The mdwtab* and mathenv† packages

Mark Wooding

28 April 1998

Contents

*The mdwtab package is currently at version 1.9, dated 28 April 1998.
†The mathenv package is currently at version 1.9, dated 28 April 1998.

List of Tables

1 User guide

The mdwtab package contains a reimplementation of the standard LaTeX tabular and array environments. This is not just an upgraded version: it's a complete rewrite. It has several advantages over the official array package (not raw LaTeX's, which is even less nice), and it's more-or-less compatible. Most of these are rather technical, I'll admit.

- The newcolumn system is properly and perfectly integrated into the system. There are now *no* 'primitive' column types – all the standard types are created as user-defined columns.
- You can define entirely different table-like environments using the equipment here. It's still hard work, although less so than before. I'll do an example of this some time.

- Construction of the preamble is generally much tidier. I've used token registers rather than `\edef`, and it's all done very nicely.
- Fine spacing before and after rules (described by DEK as 'a mark of quality') is now utterly trivial, since the preamble-generator will store the appropriate information.
- You can use array in LR and paragraph modes without having to surround it with '$' signs.
- Usually you don't want tables in the middle of paragraphs. For these cases, I've provided a simpler way to position the table horizontally.
- Footnotes work properly inside tabular environments (hoorah!). You can 'catch' footnotes using the minipage environment if you like. (It uses an internal version of the footnote package to handle footnotes, which doesn't provide extra goodies like the footnote environment; you'll need to load the full package explicitly to get them.)
- Standard LaTeX tabular environments have a problem with lining up ruled tables. The `\firsthline` command given in the *LaTeX Companion* helps a bit, but it's not really good enough, and besides, it doesn't *actually* line the text up right after all. The mdwtab package does the job properly to begin with, so you don't need to worry.

I've tested the following packages with mdwtab, and they all work. Some of the contortions required to make them work weren't pleasant, but you don't need to know about them. By a strange coincidence, all the packages were written by David Carlisle. Anyway, here's the list:

- The quite nice dcolumn package.
- The more useful delarray package.
- The rather spiffy hhline package.
- The truly wonderful tabularx package.
- The utterly magnificent longtable package.

Note that I've looked at supertabular as well: it won't work, so use longtable instead, 'cos it's much better.

1.1 The downside

There's no such thing as a free lunch. The mdwtab environment is not 100% compatible with the tabular environment found in LaTeX 2_ε or the array package.

The differences between mdwtab and LaTeX 2_ε's tabular environment are as follows:

- The vertical spacing in array environments is different to that in tabular environments. This produces more attractive results in most mathematical uses of arrays, in the author's opinion. The spacing can be modified by playing with length parameters.

- The presence of horizontal and vertical rules will alter the spacing of the table (so a pair of columns separated by a '`|`' is wider than a pair with no separation by `\arrayrulewidth`. This does mean that horizontal and vertical rules match up properly – the usual LaTeX environment makes the horizontal rules stop just short of the edge of the table, making an ugly mess (check out the *LaTeX book* if you don't believe me – page 62 provides a good example). The array package handles rules in the same way as mdwtab.

- In common with the array package, there are some restrictions on the use of the `\extracolsep` command in preambles: you may use at most one `\extracolsep` command in each '`@`' or '`!`' expression. Also, you can't say

  ```
  \newcommand{\xcs}{\extracolsep{\fill}}
  ```

  and then expect something like '`...@{\xcs}...`' to actually work – the `\extracolsep` mustn't be hidden inside any other commands. Because things like '`@`' expressions aren't expanded at the time, `\extracolsep` has to be searched and processed 'by hand'.[1]

- Control sequences (commands) in a table's preamble aren't expanded before the preamble is read. In fact, commands in the preamble are considered to be column types, and their names are entirely independent of normal LaTeX commands. No column types of this nature have yet been defined[2] but the possibility's always there. Use the `\newcolumntype` or `\coldef` commands to define new column types.

- The preamble parsing works in a completely different way. There is a certain amount of compatibility provided, although it's heavily geared towards keeping longtable happy and probably won't work with other packages.

- Obscure constructs which were allowed by the old preamble parser but violate the syntax shown in the next section (e.g., '`|@{}|`' to suppress the `\doublerulesep` space between two vertical rules, described in *The LaTeX Companion* as 'a misuse of the '`@{...}`' qualifier') are now properly outlawed. You will be given an error message if you attempt to use such a construction.

- The '`*`' forms (which repeat column types) are now expanded at a different time. Previously, preambles like '`c@*{4}{{:}@}{-}c`' were considered valid (the example would expand to '`c@{:}@{:}@{:}@{:}@{-}c`'), because '`*`'s were expanded before the preamble was actually parsed. In the new system, '`*`' is treated just like any other preamble character (it just has a rather odd action), and preambles like this will result in an error (and probably a rather confusing one).

There are also several incompatibilities between mdwtab and array:

- Because of the way `\newcolumntype` works in the array package, a horrid construction like

[1] All `\extracolsep` does is modify the `\tabskip` glue, so if you were an evil TeX hacker like me, you could just say '`\def\xcs{\tabskip=\fill}`' and put '`...@{\span\xcs}...`' in your preamble. That'd work nicely. It also works with the array package.

[2] There used to be an internal `\@magic` type used by eqnarray, but you're not supposed to know about that. Besides, it's not there any more.

```
\newcolumntype{x}{{:}}
\begin{tabular}{|c!xc|}
```

is considered to be valid, and is interpreted as '`|c!{:}c|`'. My reading of pages 54 and 55 of the *LaTeX book* tells me that this sort of thing is forbidden in normal LaTeX commands. The mdwtab preamble parser now treats column type letters much more like commands with the result that the hacking above won't work any more. The construction above would actually be interpreted as '`|c!{x}c|`' (i.e., the '`x`' column type wouldn't be expanded to '`{:}`' because the parser noticed that it was the argument to the '`!`' modifier[3]).

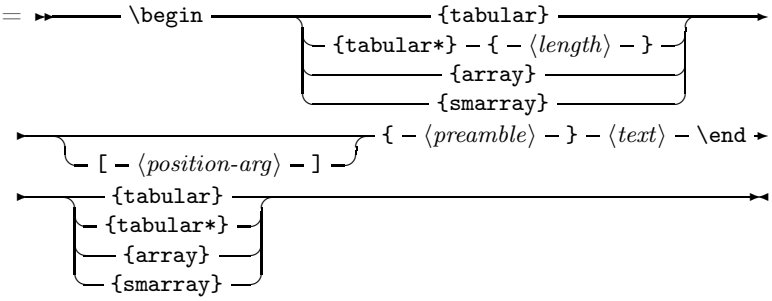
- Most of the points above, particularly those relating to the handling of the preamble, also apply to the array package. it's not such an advance over the LaTeX 2_ε version as everyone said it was.

1.2 Syntax

`tabular`
`tabular*`
`array`

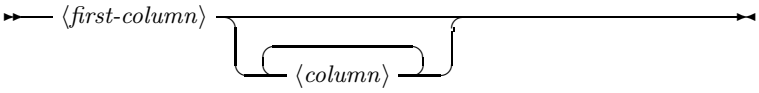
So that everyone knows where I stand, here's a complete syntax for my version of the tabular environment, and friends

⟨*tabular-env*⟩ ::=

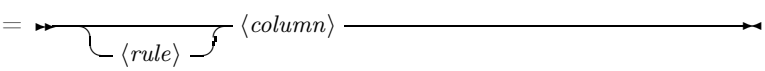


⟨*position-arg*⟩ ::= (see below)

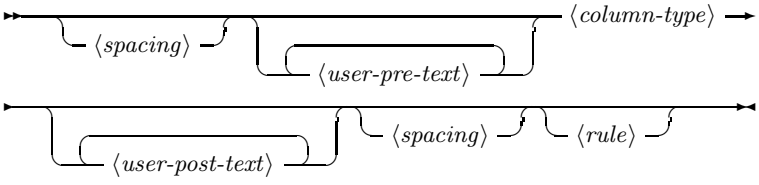
⟨*preamble*⟩ ::=



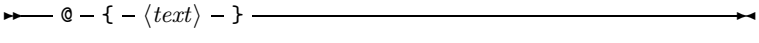
⟨*first-column*⟩ ::=



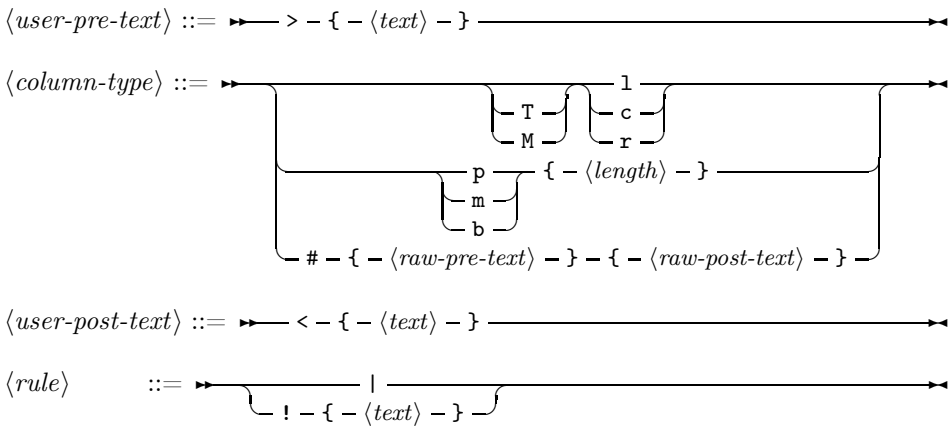
⟨*column*⟩ ::=



⟨*spacing*⟩ ::=



[3] This is a direct result of the way TeX treats undelimited arguments. See chapters 5 and 20 of *The TeXbook* for more information about how grouping affects argument reading.



If you examine the above very carefully, you'll notice a slight deviation from the original – an `@`-expression *following* a rule is considered to be part of the *next* column, not the current one. This is, I think, an almost insignificant change, and essential for some of the new features. You'll also notice the new `#` column type form, which allows you to define new real column types instead of just modifying existing ones. It's not intended for direct use in preambles – it's there mainly for the benefit of people who know what they're doing and insist on using `\newcolumntype` anyway.

The actual column types are shown in table 1.

Now that's sorted everything out, there shouldn't be any arguments at all about what a column means.

The lowercase ⟨*position-arg*⟩s '`t`', '`c`' and '`b`' do exactly what they did before: control the vertical positioning of the table. The uppercase ones control the *horizontal* positioning – this is how you create *unboxed* tables. You can only create unboxed tables in paragraph mode.

Note that unboxed tables still can't be broken across pages. Use the longtable package for this, because it already does an excellent job.

`\tabpause` One thing you can to with unboxed tables, however, is to 'interrupt' them, do some normal typesetting, and then continue. This is achieved by the `\tabpause` command: its argument is written out in paragraph mode, and the table is continued after the argument finishes. Note that it isn't a real argument as far as commands like `\verb` are concerned – they'll work inside `\tabpause` without any problems.

`\vline` The `\vline` command draws a vertical rule the height of the current table cell (unless the current cell is being typeset in paragraph mode – it only works in the simple LR-mode table cells, or in '`@`' or '`!`' modifiers). It's now been given an optional argument which gives the width of the rule to draw:

Column types	
Name	**Meaning**
`l`	Left aligned text (tabular) or equation (array).
`c`	Centred text (tabular) or equation (array).
`r`	Right aligned text (tabular) or equation (array).
`Ml`, `Mc` and `Mr`	Left, centre and right aligned equations.*
`Tl`, `Tc` and `Tr`	Left, centre and right aligned text.*
`p{`⟨*width*⟩`}`	Top aligned paragraph with the given width.
`m{`⟨*width*⟩`}`	Vertically centred paragraph with the given width.
`b{`⟨*width*⟩`}`	Bottom aligned paragraph with the given width.
`#{`⟨*pre*⟩`}{`⟨*post*⟩`}`	User defined column type: ⟨*pre*⟩ is inserted before the cell entry, ⟨*post*⟩ is inserted afterwards.*

Other modifier characters	
Name	**Meaning**
`|`	Inserts a vertical rule between columns.
`!{`⟨*text*⟩`}`	Inserts ⟨*text*⟩ between columns, treating it as a vertical rule.
`@{`⟨*text*⟩`}`	Inserts ⟨*text*⟩ instead of the usual intercolumn space.
`>{`⟨*text*⟩`}`	Inserts ⟨*text*⟩ just before the actual column entry.
`<{`⟨*text*⟩`}`	Inserts ⟨*text*⟩ just after the actual column entry.
`*{`⟨*count*⟩`}{`⟨*chars*⟩`}`	Inserts ⟨*count*⟩ copies of the ⟨*chars*⟩ into the preamble.

* This column type is a new feature

Table 1: array and tabular column types and modifiers

An example of `\vline`

A	*B*	C
D	*E*	F

```
\large
\begin{tabular}
  {| c !{\vline[2pt]} c | c |}
  \hlx{hv}
  \bf A & \it B & \sf C \\
  \hlx{vhv}
  \bf D & \it E & \sf F \\
  \hlx{vh}
\end{tabular}
```

`smarray` You've probably noticed that there's an unfamiliar environment mentioned in the syntax shown above. The smarray environment produces a 'small' array, with script size cells rather than the normal full text size cells. I've seen examples

of this sort of construction[4] being implemented by totally unsuitable commands. Someone may find it handy.

1.3 An updated \cline command

\cline The standard LaTeX \cline command has been updated. As well as just passing a range of columns to draw lines through, you can now pass a comma separated list of column numbers and ranges:

⟨*cline-cmd*⟩ ::= \cline { ⟨*number*⟩ (, ⟨*number*⟩ | - ⟨*number*⟩) }

The positioning of the horizontal lines has also been improved a bit, so that they meet up with the vertical lines properly. Displays like the one in the example below don't look good unless this has been done properly.

A \cline example

one	two	three	four
five	six	seven	eight

```
\newcommand{\mc}{\multicolumn{1}}
\begin{tabular}[C]{|c|c|c|c|}            \cline{2,4}
  \mc{c|}{one} & two & three & four   \\ \hline
  five & six & seven & \mc{c}{eight}  \\ \cline{1,3}
\end{tabular}
```

1.4 Spacing control

One of the most irritating things about LaTeX's tables is that there isn't enough space around horizontal rules. Donald Knuth, in *The TeXbook*, describes addition of some extra vertical space here as 'a mark of quality', and since TeX was designed to produce 'beautiful documents' it seems a shame that LaTeX doesn't allow this to be done nicely. Well, it does now.

\vgap The extra vertical space is added using a command \vgap, with the following syntax:

⟨*vgap-cmd*⟩ ::= \vgap ([⟨*which-cols*⟩]) { ⟨*length*⟩ }

⟨*which-cols*⟩ ::= (⟨*number*⟩ (, ⟨*number*⟩ | - ⟨*number*⟩))

This command must appear either immediately after the beginning of the table or immediately after the \\ which ends a row. (Actually, there are other commands

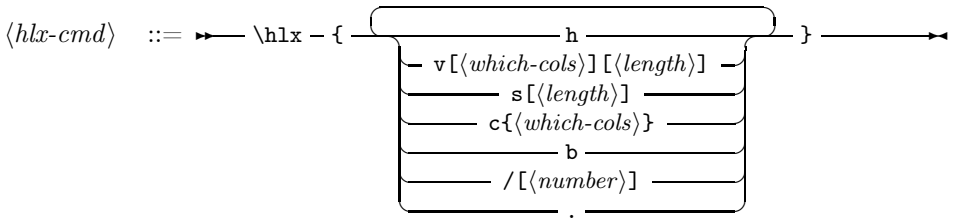
[4] There's a nasty use of smallmatrix in the testmath.tex file which comes with the amslatex distribution. It's actually there to simulate a 'smallcases' environment, which the mathenv package includes, based around smarray.

which also have this requirement – you can specify a collection of them wherever you're allowed to give any one.) It adds some vertical space (the amount is given by the ⟨*length*⟩) to the table, making sure that the vertical rules of the table are extended correctly.

The `\vgap` command relies on information stored while your table preamble is being examined. However, it's possible that you might not want some of the rules drawn (e.g., if you've used `\multicolumn`). The optional ⟨*which-cols*⟩ argument allows you to specify which rules are *not* to be drawn. You can specify either single column numbers or ranges. The rule at the very left hand side is given the number 0; the rules at the end of column n are numbered n. It's easy really.

`\hlx` Using `\vgap` is all very well, but it's a bit cumbersome, and takes up a lot of typing, especially when combined with `\hline` commands. The `\hlx` command tries to tidy things.

The syntax is simple:



The argument works a bit like a table preamble, really. Each letter is a command. The following are supported:

`h` Works just like `\hline`. If you put two adjacent to each other, a gap will be put between them.

`v[`⟨*which-cols*⟩`][`⟨*length*⟩`]` Works like `\vgap[`⟨*which-cols*⟩`]{`⟨*length*⟩`}`. If the ⟨*length*⟩ is omitted, the value of `\doublerulesep` is used. This usually looks right.

`s[`⟨*length*⟩`]` Leaves a vertical gap with the given size. If you omit the ⟨*length*⟩ then `\doublerulesep` is used. This is usually right.

`c{`⟨*which-cols*⟩`}` Works just like `\cline`.

`b` Inserts a backspace the width of a rule. This is useful when doing longtables.

`/[`⟨*number*⟩`]` Allows a page break in a table. Don't use this except in a longtable environment. The ⟨*number*⟩ works exactly the same as it does in the `\pagebreak` command, except that the default is 0, which just permits a break without forcing it.

`.` (That's a dot) Starts the next row of the table. No more characters may follow the dot, and no `\hline`, `\hlx`, `\vgap` or `\multicolumn` commands may be used after it. You don't have to include it, and most of the time it's totally useless. It can be handy for some macros, though. I used it in (and in fact added it especially for) the table of column types.

An example of the use of `\hlx` is given, so you can see what's going on.

Beautiful table example

AT&T Common Stock		
Year	**Price**	**Dividend**
1971	41–54	$2.60
2	41–54	2.70
3	46–55	2.87
4	40–53	3.24
5	45–52	3.40
6	51–59	.95*

* (first quarter only)

```
\newcommand{\zerowidth}[1]{\hbox to 0pt{\hss#1\hss}}
\setlength{\tabcolsep}{1.5em}
\begin{tabular}[C]{| r | c | r |}                    \hlx{hv[1,2]}
  \multicolumn{3}{|c|}{\bf AT\&T Common Stock}  \\ \hlx{v[1,2]hv}
  \multicolumn{1}{|c|}{\zerowidth{\bf Year}} &
  \multicolumn{1}{c|}{\zerowidth{\bf Price}} &
  \multicolumn{1}{c|}{\zerowidth{\bf Dividend}} \\ \hlx{vhv}
  1971 & 41--54 & \$2.60                          \\
     2 & 41--54 &   2.70                          \\
     3 & 46--55 &   2.87                          \\
     4 & 40--53 &   3.24                          \\
     5 & 45--52 &   3.40                          \\
     6 & 51--59 &    .95\rlap{*}                  \\ \hlx{vhs}
  \multicolumn{3}{@{}l}{* (first quarter only)}
\end{tabular}
```

1.5 Creating beautiful long tables

You can use the `\vgap` and `\hlx` commands with David Carlisle's stunning longtable package. However, there are some things you should be away of to ensure that your tables always come out looking lovely.

The longtable package will break a table at an `\hline` command, leaving a rule at the bottom of the page and another at the top of the next page. This means that a constructions like `\hlx{vhv}` will be broken into something like `\hlx{vh}` at the bottom of the page and `\hlx{hv}` at the top of the next. You need to design the table headers and footers with this in mind.

However, there appears to be a slight problem:[5] if the footer starts with an `\hline`, and a page is broken at an `\hline`, then you get an extra thick rule at the bottom of the page. This is a bit of a problem, because if the rule isn't there in the footer and you get a break between two rows *without* a rule between them, then the page looks very odd.

[5] You might very well call it a bug. I couldn't possibly comment.

If you want to do ruled longtables, I'd recommend that you proceed as follows:

- End header sections with an `\hlx{vh}`.
- Begin footer sections with an `\hlx{bh}`.
- Begin the main table with `\hlx{v}`.
- Insert `\hlx{vhv}` commands in the main table body as usual.

If longtable gets modified appropriately, the use of the '`b`' command won't be necessary.

Here's an example of the sort of thing you'd type.

```
\begin{longtable}[c]{|c|l|}          \hlx{hv}
\bf Heading & \bf Also heading     \\ \hlx{vh}
\endhead
\hlx{bh}
\endfoot
\hlx{v}
First main & table line            \\ \hlx{vhv}
Lots of text & like this           \\ \hlx{vhv}
  ⋮
Lots of text & like this           \\ \hlx{vhv}
Last main & table line            \\ \hlx{vh}
\end{longtable}
```

1.6 Rules and vertical positioning

In the LaTeX 2_ε and array.sty versions of tabular, you run into problems if you try to use ruled tables together with the '`[t]`' or '`[b]`' position specifiers – the top or bottom rule ends up being nicely lined up with the text baseline, giving you an effect which is nothing like the one you expected. The *LaTeX Companion* gives two commands `\firsthline` and `\lasthline` which are supposed to help with this problem. (These commands have since migrated into the array package.) Unfortunately, `\firsthline` doesn't do its job properly – it gets the text position wrong by exactly the width of the table rules.

The mdwtab package makes all of this automatic. It gets the baseline positions exactly right, whether or not you use rules. Earlier versions of this package required that you play with a length parameter called `\rulefudge`; this is no longer necessary (or even possible – the length parameter no longer exists). The package now correctly compensates for all sorts of rules and `\vgap`s at the top and bottom of a table and it gets the positioning right all by itself. You've never had it so good.

1.7 User serviceable parts

There are a lot of parameters which you can modify in order to make arrays and tables look nicer. They are all listed in table 2.

Parameter	Meaning
\tabstyle	A command executed at the beginning of a tabular or tabular∗ environment. By default does nothing. Change using \renewcommand.
\extrarowheight	A length added to the height of every row, used to stop table rules overprinting ascenders. Default 0 pt. Usage is deprecated now: use \hlx instead.
\tabextrasep	Extra space added between rows in a tabular or tabular∗ environment (added *before* any following \hline). Default 0 pt.
\arrayextrasep	Analogous to \tabextrasep, but for array environments. Default 1 jot (3 pt).
\smarrayextrasep	Analogous to \tabextrasep, but for smarray environments. Default 1 pt.
\tabcolsep	Space added by default on each side of a table cell (unless suppressed by an '@'-expression) in tabular environments. Default is defined by your document class.
\arraycolsep	Analogous to \tabcolsep, but for array environments. Default is defined by your document class.
\smarraycolsep	Analogous to \tabcolsep, but for smarray environments. Default is 3 pt.
\arrayrulewidth	The width of horizontal and vertical rules in tables.
\doublerulesep	Space added between two adjacent vertical or horizontal rules. Also used by \hlx{v}.
\arraystretch	Command containing a factor to multiply the default row height. Default is defined by your document class (usually 1).

Table 2: Parameters for configuring table environments

1.8 Defining column types

\newcolumntype The easy way to define new column types is using \newcolumntype. It works in more or less the same way as \newcommand:

⟨*new-col-type-cmd*⟩ ::= ▸▸ \newcolumntype { ⟨*column-name*⟩ } → { → [⟨*num-args*⟩] [⟨*default-arg*⟩] → ⟨*first-column*⟩ } → ⟨*column*⟩

(The array.sty implementation doesn't accept the ⟨*default-arg*⟩ argument. I've no idea why not, 'cos it was very easy to implement.)

\colset This implementation allows you to define lots of different sets of columns. You can change the current set using the \colset declaration:

⟨*colset-cmd*⟩ ::= ▸▸ \colset { ⟨*set-name*⟩ } →

This leaves a problem, though: at any particular moment, the current column set could be anything, since other macros and packages can change it.

\colpush
\colpop

What actually happens is that a stack of column sets is maintained. The `\colset` command just replaces the item at the top of the stack. The command `\colpush` pushes its argument onto the top of the stack, making it the new current set. The corresponding `\colpop` macro (which doesn't take any arguments) removes the top item from the stack, reinstating the previous current column set.

⟨*colpush-cmd*⟩ ::= `\colpush` – `{` – ⟨*set-name*⟩ – `}`

⟨*colpop-cmd*⟩ ::= `\colpop`

The macros which manipulate the column set stack work *locally*. The contents of the stack are saved when you open a new group.

To make sure everyone behaves themselves properly, these are the rules for using the column set stack:

- Packages defining column types must ensure that they preserve the current column set. Either they must push their own column type and pop it off when they're finished defining columns, or they must avoid changing the stack at all, and use the optional arguments to `\coldef` and `\collet`.
- Packages must not assume that any particular column set is current unless they have made sure of it themselves.
- Packages must ensure that they pop exactly as much as they push. There isn't much policing of this (perhaps there should be more), so authors are encouraged to behave responsibly.
- Packages must change the current column set (using `\colset`) when they start up their table environment. This will be restored when the environment closes.

\coldef

`\newcolumntype` is probably enough for most purposes. However, Real TEXnicians, and people writing new table-generating environments, require something lower-level.

⟨*coldef-cmd*⟩ ::= `\coldef` [– ⟨*set-name*⟩ – `]`] – ⟨*col-name*⟩ – ⟨*arg-template*⟩ – `{` – ⟨*replacement-text*⟩ – `}`

Note that this defines a column type in the current colset. It works almost exactly the same way as TEX's primitive `\def`. There is a potential gotcha here: a `\tab@mkpream` token is inserted at the end of your replacement text. If you need to read an optional argument or something, you'll need to gobble this token before you carry on. The `\@firstoftwo` macro could be handy here:

```
\coldef x{\@firstoftwo{\@ifnextchar[\@xcolumn@i\@xcolumn@ii}}}
```

This isn't a terribly pretty state of affairs, and I ought to do something about it. I've not seen any use for an optional argument yet, though. Note that if you do gobble the `\tab@mkpream`, it's your responsibility to insert another one at the very end of your macro's expansion (so that further preamble characters can be read).

The replacement text is inserted directly. It's normal to insert preamble elements here. There are several to choose from:

Column items provide the main 'meat' of a column. You insert a column element by saying `\tabcoltype{`⟨*pre-text*⟩`}{`⟨*post-text*⟩`}`. The user's text gets inserted between these two. (So do user pre- and post-texts. Bear this in mind.)

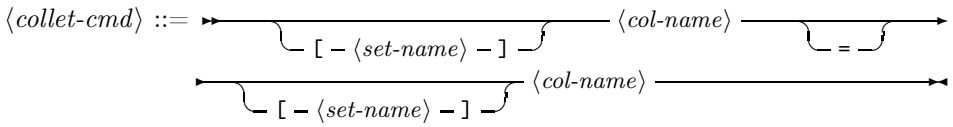
User pre-text items work like the '`>`' preamble command. You use the `\tabuserpretype{`⟨*text*⟩`}` command to insert it. User pre-texts are written in *reverse* order between the pre-text of the column item and the text from the table cell.

User post-text items work like the '`<`' preamble command. You use the `\tabuserposttype{`⟨*text*⟩`}` command to insert it. Like user pre-texts, user post-texts are written in reverse order, between the table cell text and the column item post-text.

Space items work like the '`@`' preamble command. They're inserted with the `\tabspctype{`⟨*text*⟩`}` command.

Rule items work like the '`|`' and '`!`' commands. You insert them with the `\tabruletype{`⟨*text*⟩`}` command. Note that the text is inserted by `\vgap` too, so it should contain things which adjust their vertical size nicely. If you really need to, you can test `\iftab@vgap` to see if you're in a `\vgap`.

`\collet` As well as defining columns, you can copy definitions (rather like `\let` allows you to copy macros). The syntax is like this:



(In other words, you can copy defintions from other column sets.)

1.9 Defining new table-generating environments

Quite a few routines are provided specifically to help you to define new environments which do alignment in a nice way.

1.9.1 Reading preambles

The main tricky bit in doing table-like environments is parsing preambles. No longer.

`\tab@readpreamble` `\tab@doreadpream` The main parser routine is called `\tab@doreadpream`. Given a user preamble string as an argument, it will build an `\halign` preamble to return to you. However, the preamble produced won't be complete. This is because you can actually make multiple calls to `\tab@doreadpream` with bits of user preambles. The `\newcolumntype` system uses this mechanism, as does the '`*`' (repeating) modifier. When there really is no more preamble to read, you need to *commit* the heldover tokens to the output. The `\tab@readpreamble` routine will do this for you – given a user preamble, it builds a complete output from it.

A token register `\tab@preamble` is used to store the generated preamble. Before starting, you must iniitialise this token list to whatever you want. There's

another token register, \tab@shortline, which is used to store tokens used by \vgap. For each column in the table, the list contains an \omit (to override the standard preamble) and an \hfil space taking up most of the column. Finally, for each rule item in the user preamble, the shortline list contains an entry of the form:

\tab@ckr{⟨*column-number*⟩}{⟨*rule-text*⟩}

This is used to decide whether to print the rule or an empty thing of the same width. You probably ought to know that the very first column does *not* have a leading \omit – this is supplied by \vgap so that it can then look for optional arguments.

\tab@initread As well as initialising \tab@preamble and emptying \tab@shortline, there are several other operations required to initialise a preamble read. These are all performed by the \tab@initread macro, although you may want to change some of the values for your specific application. For reference, the actions performed are:

- initialising the parser state by setting \tab@state = \tab@startstate;
- clearing the token lists \tab@preamble and \tab@shortlist;
- initialising the macros \tab@tabtext, \tab@midtext, and \tab@multicol to their default values of '&', '\ignorespaces#\unskip' and the empty token list respectively.[6]
- clearing the internal token list registers \tab@pretext, tab@userpretext and \tab@posttext;
- clearing the column counter \tab@columns to zero;
- clearing the action performed when a new column is started (by making the \tab@looped macro equal to \relax; this is used to make \multicolumn macro raise an error if you try to do more than one column); and
- setting up some other switches used by the parser (\iftab@rule, \iftab@initrule and \iftab@firstcol, all of which are set to be true).

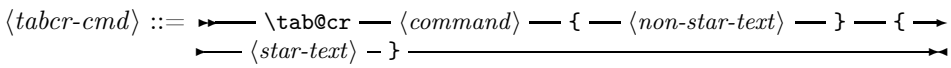
The macro \tab@multicol is used by the \multicolumn command to insert any necessary items (e.g., struts) before the actual column text. If you set this to something non-empty, you should probably consider adding a call to the macro to the beginning of \tab@preamble.

When parsing is finally done, the count register \tab@columns contains the number of columns in the alignment. Don't corrupt this value, because it's used for handling \hline commands.

1.9.2 Starting new lines

The other messy bit required by table environments is the newline command \\. There are nasty complications involved with starting new lines, some of which can be handled by this package, and some on which I can only give advice.

\tab@cr The optional arguments and star-forms etc. can be read fairly painlessly using the \tab@cr command:



This will call your ⟨*command*⟩ with two arguments. The first is the contents of the optional argument, or '\z@' if there wasn't one. The second is either ⟨*star-text*⟩ or ⟨*non-star-text*⟩ depending on whether the user wrote the ∗-form or not.

Somewhere in your ⟨*command*⟩, you'll have to use the \cr primitive to end the table row. After you've done this, you *must* ensure that you don't do anything that gets past TEX's mouth without protecting it – otherwise \hline and co. won't work. I usually wrap things up in a \noalign to protect them, although there are other methods. Maybe.

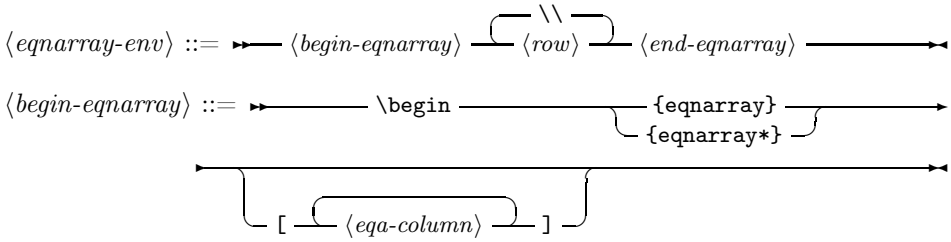
You might like to have a look at the eqnarray implementation provided to see how all this gets put into practice.

1.10 The mathenv package alignment environments

The mathenv package provides several environments for aligning equations in various ways. They're mainly provided as a demonstration of the table handling macros in mdwtab, so don't expect great things. If you want truly beautiful mathematics, use amsmath.[7] However, the various environments do nest in an approximately useful way. I also think that the matrix and script environments provided here give better results than their amsmath equivalents, and they are certainly more versatile.

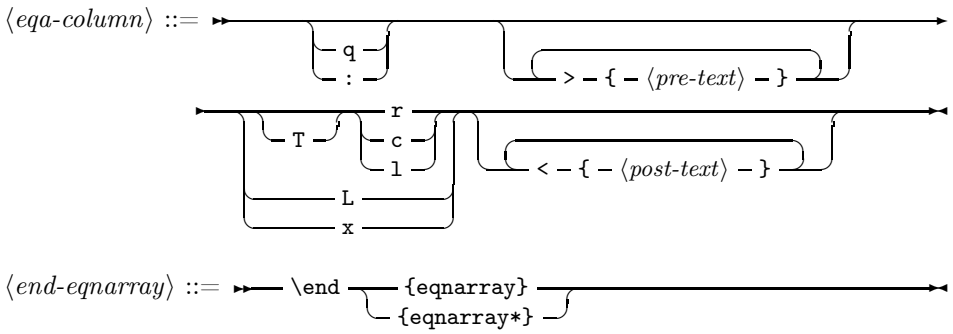
1.10.1 The new eqnarray environment

eqnarray
eqnarray* As an example of the new column defining features, and because the original isn't terribly good, I've included a rewritten version of the eqnarray environment. The new implementation closes the gap between eqnarray and $\mathcal{AMS}$-TEX alignment features. It's in a separate, package called mathenv, to avoid wasting your memory.



[6] These are macros rather than token lists to avoid hogging all the token list registers. Actually, the package only allocates two, although it does use almost all of the temporary registers as well. Also, there's a lie: \unskip is too hamfisted to remove trailing spaces properly; I really use a macro called \@maybe@unskip

[7] Particularly since nice commands like \over are being reactivated in a later release of amsmath.



Descriptions of the various column types are given in table 3.

Column types	
Name	**Meaning**
`l`	Left aligned piece of equation.
`c`	Centred piece of equation.
`x`	Centred or flush-left whole equation (depending on fleqn option).
`r`	Right aligned piece of equation.
`L`	Left aligned piece of equation whose width is considered to be 2 em.
`Tl`, `Tc` and `Tr`	Left, centre and right aligned text.
Other modifier characters	
Name	**Meaning**
`:`	Leaves a big gap between equations. By default, the 'chunks' separated by ':'s are equally spaced on the line.
`q`	Inserts 1 em of space
`>{`⟨*text*⟩`}`	Inserts ⟨*text*⟩ just before the actual column entry.
`<{`⟨*text*⟩`}`	Inserts ⟨*text*⟩ just after the actual column entry.
`*{`⟨*count*⟩`}{`⟨*chars*⟩`}`	Inserts ⟨*count*⟩ copies of the ⟨*chars*⟩ into the preamble.

Table 3: eqnarray column types and modifiers

The default preamble, if you don't supply one of your own, is '`rcl`'. Most of the time, '`rl`' is sufficient, although compatibility is more important to me.

By default, there is no space between columns, which makes formulæ in an eqnarray environment look just like formulæ typeset on their own, except that things get aligned in columns. This is where the default eqnarray falls down: it leaves `\arraycolsep` space between each column making the thing look horrible.

An example would be good here, I think. This one's from exercise 22.9 of the *TEXbook*.

Simultaneous equations

$$\begin{aligned} 10w + \ 3x + 3y + 18z &= 1 \qquad (1)\\ 6w - 17x \quad - \ 5z &= 2 \qquad (2) \end{aligned}$$

```
\begin{eqnarray}[*3{rc}rl]
  10w & + &  3x & + & 3y & + & 18z & = 1 \\
   6w & - & 17x &   &    & - &  5z & = 2
\end{eqnarray}
```

Choosing a more up-to-date example, here's some examples from the *LaTeX Companion*.

Lots of equations

$$\begin{aligned} V_i &= v_i - q_i v_j, \quad & X_i &= x_i - q_i x_j, \quad & U_i &= u_i, \quad \text{for } i \neq j \qquad (3)\\ V_j &= v_j, & X_j &= x_j & U_j u_j &+ \sum_{i \neq j} q_i u_i. \qquad (4) \end{aligned}$$

```
\begin{eqnarray}[rl:rl:lq]
 V_i &= v_i - q_i v_j, & X_i &= x_i - q_i x_j, &
       U_i = u_i, \qquad \mbox{for $i \ne j$}  \\
 V_j &= v_j,           & X_j &= x_j            &
       U_j u_j + \sum_{i \ne j} q_i u_i. \label{eq:A}
\end{eqnarray}
```

Plain text column and `\tabpause`

$$\begin{aligned} x &= y && \text{by (4)} \qquad (5)\\ x' &= y' && \text{by definition} \qquad (6) \end{aligned}$$

and

$$x + x' = y + y' \qquad \text{by Axiom 1} \qquad (7)$$

```
\begin{eqnarray}[rlqqTl]
     x  &= y           & by (\ref{eq:A}) \\
     x' &= y'          & by definition \\
\tabpause{and}
 x + x' &= y + y'      & by Axiom~1
\end{eqnarray}
```

The new features also mean that you don't need to mess about with `\lefteqn` any more. This is handled by the 'L' column type:

Splitting example

$$w + x + y + z = \\ a + b + c + d + e + \\ f + g + h + i + j$$

```
\begin{eqnarray*}[Ll]
   w+x+y+z = \\
    & a+b+c+d+e+ \\
    & f+g+h+i+j
\end{eqnarray*}
```

Finally, just to prove that the spacing's right at last, here's another one from the *Companion*.

Spacing demonstration

$$x^2 + y^2 = z^2 \quad (8)$$
$$x^2 + y^2 = z^2 \quad (9)$$
$$y^2 < z^2 \quad (10)$$

```
\begin{equation}
  x^2 + y^2 = z^2
\end{equation}
\begin{eqnarray}[rl]
  x^2 + y^2 &= z^2 \\
        y^2 &< z^2
\end{eqnarray}
```

Well, that was easy enough. Now on to numbering. As you've noticed, the equations above are numbered. You can use the eqnarray∗ environment to turn off the numbering in the whole environment, or say `\nonumber` on a line to suppress numbering of that one in particular.

`\eqnumber` More excitingly, you can say `\eqnumber` to enable numbering for a particular equation, or `\eqnumber[`⟨*text*⟩`]` to choose what to show instead of the line number. This works for both starred and unstarred versions of the environment. Now `\nonumber` becomes merely a synonym for '`\eqnumber[]`'.

A note for cheats: you can use the sparkly new eqnarray for simple equations by specifying '`x`' as the column description. Who needs $\mathcal{AMS}$-TEX? `;-)`

`eqlines` `eqlines*` In fact, there's a separate environment eqlines, which is equivalent to eqnarray with a single '`x`' column; the result is that you can insert a collection of displayed equations separated by `\\` commands. If you don't like numbering, use eqlines∗ insead.

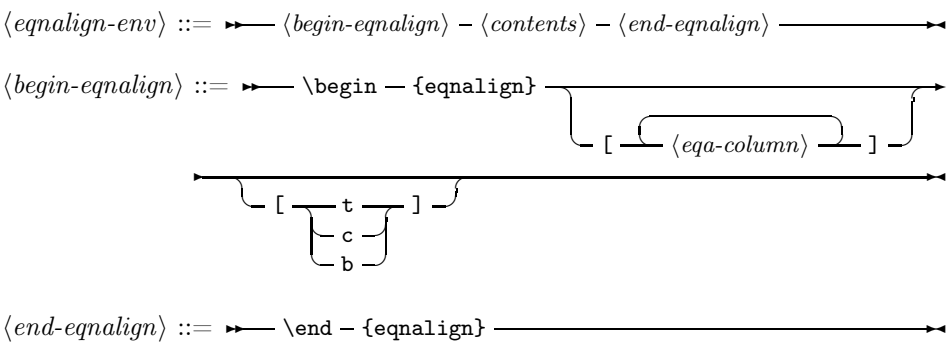
1.10.2 The eqnalign environment

`eqnalign` There's a new environment, eqnalign, which does almost the same thing as eqnarray but not quite. It doesn't do equation numbers, and it wraps its contents up in a box. The result of this is that:

- You can use eqnalign for just a part of a formula. The eqnarray environment must take up the whole display.
- You can use eqnalign within eqnarray for extra fine alignment of subsidiary bits.

- You can break off from doing an eqnarray using the `\tabpause` command. You can't use `\tabpause` inside eqnalign.[8]

The eqnalign environment works like this:



As the syntax suggests, the preamble for the eqnalign environment works exactly the same way as for eqnarray. Example time: another one from the *TEXbook*.

Example of eqnalign

$$\left\{\begin{aligned}\alpha &= f(z)\\ \beta &= f(z^2)\\ \gamma &= f(z^3)\end{aligned}\right\} \qquad \left\{\begin{aligned}x &= \alpha^2 - \beta\\ y &= 2\gamma\end{aligned}\right\}.$$

```
\[
  \left\{ \begin{eqnalign}[rl]
    \alpha &= f(z) \\ \beta  &= f(z^2) \\
    \gamma &= f(z^3)
  \end{eqnalign} \right\}
  \qquad
  \left\{ \begin{eqnalign}[rl]
    x &= \alpha^2 - \beta \\ y &= 2\gamma
  \end{eqnalign} \right\}.
\]
```

`\multicolumn` The `\multicolumn` command works correctly in both the eqnarray and eqnalign environments, although you should bear in mind that you should give eqnarray column types, not array ones.

1.10.3 A note on spacing in alignment environments

Most of the time, equations in eqnarray and eqnalign environments will be beautiful. However, there are some things you should bear in mind when you produce beautiful equations.

[8] Well, technically speaking there's nothing to stop you. However, the results won't be pretty.

The main problem with spacing is making sure that binary relations and binary operators have the correct amount of space on each side of them. The alignment environments insert 'hidden' objects at the ends of table cells to assist with the spacing: '`l`' column types have a hidden object on the left, '`r`' types have a hidden object on the right, and '`c`' types have a hidden object on *both* ends. These hidden objects add the correct space when there's a binary operator or relation next to them. If some other sort of object is lurking there, no space is added. So far, so good.

The only problem comes when you have something like this:

How not to do an eqnarray

$$\begin{aligned} x + y &= 12 \\ 2x - 5y &= -6 \end{aligned}$$

```
\begin{eqnarray*}[rcl]
  x +  y & = & 12 \\
 2x - 5y & = & -6
\end{eqnarray*}
```

The '$-$' sign in the second equation has been treated as a binary operator when really it should be a unary prefix operator, but TEX isn't clever enough to know the difference. (Can you see the difference in the spacing between -6 and -6?) There are two possible solutions to the problem. You could wrap the '`-6`' up in a group ('`{-6}`'), or just the $-$ sign ('`{-}6`'). A better plan, though, is to get rid of the middle column altogether:

How to do an eqnarray

$$\begin{aligned} x + y &= 12 \\ 2x - 5y &= -6 \end{aligned}$$

```
\begin{eqnarray*}[rl]
  x +  y & = 12 \\
 2x - 5y & = -6
\end{eqnarray*}
```

Since the things in the middle column were the same width, it's not actually doing any good. Also, now that TEX can see that the thing on the left of the '$-$' sign is a relation (the '$=$' sign), it will space the formula correctly.

In this case, it might be even better to add some extra columns, and line up the x and y terms in the left hand side:

Extra beautiful eqnarray

$$\begin{aligned} x + \ y &= 12 \\ 2x - 5y &= -6 \end{aligned}$$

```
\begin{eqnarray*}[rrl]
  x + &  y & = 12 \\
 2x - & 5y & = -6
\end{eqnarray*}
```

There's no need to put the '$+$' and '$-$' operators in their own column here, because they're both 7.7778 pt wide, even though they don't look it.

1.10.4 Configuring the alignment environments

There are a collection of parameters you can use to make the equation alignment environments (eqnarray and eqnalign) look the way you like them. These are all shown in table 4.

Parameter	Use
\eqaopenskip	Length put on the left of an eqnarray environment. By default, this is \@centering (to centre the alignment) or \mathindent (to left align) depending on whether you're using the fleqn document class option.
\eqacloseskip	Length put on the right of an eqnarray environment. By default, this is \@centering, to align the environment correctly.
\eqacolskip	Space added by the ':' column modifier. This should be a rubber length, although it only stretches in eqnarray, not in eqnalign. The default value is $1^1/_2$ em with 1000 pt of stretch.
\eqainskip	Space added at each side of a normal column. By default this is 0 pt.
\eqastyle	The maths style used in the alignment. By default, this is \textstyle, and you probably won't want to change it.

Table 4: Parameters for the eqnarray and eqnalign environments

1.11 Other multiline equations

Sometimes there's no sensible alignment point for splitting equations. The normal thing to do under these circumstances is to put the first line way over to the left of the page, and the last line over to the right. (If there are more lines, I imagine we put them in the middle.)

spliteqn
spliteqn*

The spliteqn environment allows you to do such splitting of equations. Rather than tediously describe it, I'll just give an example, because it's really easy. The ∗-version works the same, except it doesn't put an equation number in.

subsplit

If you have a very badly behaved equation, you might want to split a part of it (say, a bit of a fraction), particularly if you're doing things in narrow columns.

1.12 Matrices

Also included in the mathenv package is a collection of things for typesetting matrices. The standard array doesn't (in my opinion) provide the right sort of spacing for matrices. Plain TEX provides some quite nice matrix handling macros, but they don't work in the appropriate LATEX way.

Warning: These definitions will make old versions of plain.sty unhappy; newer versions correctly restore the Plain TEX macros \matrix and \pmatrix.

matrix

The simple way to do matrices is with the matrix environment.

⟨*matrix-env*⟩ ::= ⟨*begin-matrix*⟩ – ⟨*contents*⟩ – ⟨*end-matrix*⟩

⟨*begin-matrix*⟩ ::= \begin{matrix} [– ⟨*matrix-cols*⟩ –]

A split equation

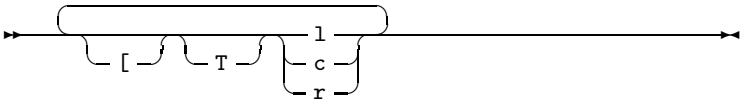
$$\sum_{1\le j\le n} \frac{1}{(x_j - x_1)\dots(x_j - x_{j-1})(x - x_j)(x_j - x_{j+1})\dots(x_j - x_n)} = \frac{1}{(x - x_1)\dots(x - x_n)}. \quad (11)$$

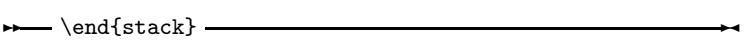
```
\begin{spliteqn}
  \sum_{1\le j\le n}
  \frac {1} { (x_j - x_1) \ldots (x_j - x_{j-1})
              (x - x_j) (x_j - x_{j+1}) \ldots (x_j - x_n) }
  \\
  = \frac {1} { (x - x_1) \ldots (x - x_n) }.
\end{spliteqn}
```

A subsplit environment

$$\frac{q^{\frac{1}{2}n(n+1)}(ea;q^2)_\infty(eq/a;q^2)_\infty \\ (caq/e;q^2)_\infty(cq^2/ae;q^2)_\infty}{(e;q)_\infty(cq/e;q)_\infty} \quad (12)$$

```
\begin{equation}
  \frac{
    \begin{subsplit}
      q^{\frac{1}{2} n(n+1)}(ea; q^2)_\infty (eq/a; q^2)_\infty \\
                  (caq/e; q^2)_\infty (cq^2/ae; q^2)_\infty
    \end{subsplit}
  }{
    (e; q)_\infty (cq/e; q)_\infty
  }
\end{equation}
```

⟨*matrix-cols*⟩ ::= 

⟨*end-matrix*⟩ ::= 

The '`l`', '`c`' and '`r`' columns are fairly obvious – they align their contents in the appropriate way. The '`[`' character is more complicated. It means 'repeat the remaining column types forever', so a preamble of '`cc[lr`' means 'two centred columns, then alternating left- and right-aligned columns for as often as needed'. The default preamble, if you don't specify one, is '`[c`' – 'any number of centred columns'.

`\multicolumn` The `\multicolumn` command works correctly in matrices, although you should bear in mind that you should give matrix column types, not array ones.

`pmatrix` The standard matrix environment doesn't put any delimiters around the matrix. You can use the standard `\left` and `\right` commands, although this is a bit nasty. The pmatrix environment will put parentheses around the matrix it creates; it's otherwise exactly the same as matrix.

`dmatrix` A dmatrix environment is also provided. It takes two extra arguments: the left and right delimiter characters (without `\left` or `\right`).

Various matrix environments

$$\begin{matrix} 1 & 0 \\ 0 & -1 \end{matrix} \quad \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \quad \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

```
\[ \begin{matrix} 1 & 0 \\ 0 & -1 \end{matrix} \quad
   \begin{pmatrix}
     \cos\theta & \sin\theta \\
    -\sin\theta & \cos\theta
   \end{pmatrix} \quad
   \begin{dmatrix}[] 0 & -i \\ i & 0 \end{dmatrix}
\]
```

`smatrix` Normal matrices always come out the same size; they don't change size according to the surrounding context (unfortunately). However, it can be occasionally useful to put matrices in running text, so you can talk about A being $\left(\begin{smallmatrix} a & b \\ b & c \end{smallmatrix}\right)$ being its own transpose (i.e., $A = A^T$). This is accomplished using the smatrix (the 's' stands for 'small' – I thought that 'smallmatrix' was too big to type inline). As well as inline text, the smatrix can be useful in displays, if the matrix is deep in a subformula. I can't think of any examples offhand, though.

`spmatrix` `sdmatrix` The smatrix environment doesn't supply any delimiters, like matrix. There are spmatrix and sdmatrix environments which do, though. Note that delimiters have a tendency to get too big and mess up the line spacing – I had to use explicitly `\big` delimiters in the above example.

pmatrix*
spmatrix*
sdmatrix*

All the small matrix environments have starred versions, which are more suitable for use in displays, since they have more space between the rows. They're intended for typesetting really big matrices in displays.

\ddots
\vdots

The standard \vdots and \ddots commands don't produce anything at all nice in small matrices, so this package redefines them so that they scale properly to smaller sizes.

genmatrix

Actually, all these environments are special cases of one: genmatrix. This takes oodles of arguments:

```
\begin{genmatrix}{⟨matrix-style⟩}{⟨outer-style⟩}
    {⟨spacing⟩}{⟨left-delim⟩}{⟨right-delim⟩}
  ⋮
\end{genmatrix}
```

The two 'style' arguments should be things like \textstyle or \scriptstyle; the first, ⟨*matrix-style*⟩, is the style to use for the matrix elements, and the second, ⟨*outer-style*⟩, is the style to assume for the surrounding text (this affects the spacing within the matrix; it should usually be the same as ⟨*matrix-style*⟩). The ⟨*spacing*⟩ is inserted between the matrix and the delimiters, on each side of the matrix. It's usually '\,' in full-size matrices, and blank for small ones. The delimiters are inserted around the matrices, and sized appropriately.

newmatrix

You can create your own matrix environments if you like, using the \newmatrix command. It takes two arguments, although they're a bit odd. The first is the name of the environment, and the second contains the arguments to pass to genmatrix. For example, the pmatrix environment was defined by saying

```
\newmatrix{pmatrix}{{\textstyle}{\textstyle}{\,}{(}{)}}
```

If you don't pass all three arguments, then you end up requiring the user to specify the remaining ones. This is how dmatrix works.

script

Finally, although it's not really a matrix, stacked super- and subscripts follow much the same sorts of spacing rules. The script environment allows you to do this sort of thing very easily. It essentially provides a 'matrix' with the right sort of spacing. The default preamble string is 'c', giving you centred scripts, although you can say \begin{script}[l] for left-aligned scripts, which is better if the script is being placed to the right of its operator. If you're really odd, you can have more than one column.

Example of script

$$\sum_{x\in A}{}' f(x) \stackrel{\mathrm{def}}{=} \sum_{\substack{x\in A\\ x\ne 0}} f(x)$$

```
\[ \mathop{{\sum}'}_{x \in A}
   f(x)
   \stackrel{\mathrm{def}}{=}
   \sum_{\begin{script}
     x \in A \\ x \ne 0
   \end{script}} f(x)
\]
```

1.13 Other mathenv environments

The mathenv package contains some other environments which may be useful, based on the enhanced tabular and array environments.

cases The cases environment lets you say things like the following:

Example of cases

$$P_{r-j} = \begin{cases} 0 & \text{if } r-j \text{ is odd} \\ r!\,(-1)^{(r-j)/2} & \text{if } r-j \text{ is even} \end{cases}$$

```
\[ P_{r-j} = \begin{cases}
               0 & if $r-j$ is odd \\
               r!\,(-1)^{(r-j)/2} & if $r-j$ is even
             \end{cases}
\]
```

The spacing required for this is a bit messy, so providing an environment for it is quite handy.

smcases The smcases environment works the same way as cases, but with scriptsize lettering.

2 Implementation of table handling

Here we go. It starts horrid and gets worse. However, it does stay nicer than the original, IMHO.

1 ⟨*mdwtab⟩

2.1 Registers, switches and things

We need lots of these. It's great fun.

The two count registers are simple enough:

\tab@state contains the current parser state. Since we probably won't be parsing preambles recursively, this is a global variable.

\tab@columns contains the number of the current column.

\tab@hlstate contains the state required for hline management.

```
2 \newcount\tab@state
3 \newcount\tab@columns
```

We need *lots* of token registers. Fortunately, most of them are only used during parsing. We'll use Plain TeX's scratch tokens for this. Note that \toks\tw@ isn't used here. It, and \toks@, are free for use by column commands.

```
4 \newtoks\tab@preamble
5 \newtoks\tab@shortline
6 \toksdef\tab@pretext 4
7 \toksdef\tab@posttext 6
8 \toksdef\tab@userpretext 8
```

The dimens are fairly straightforward. The inclusion of `\col@sep` is a sacrifice to compatibility – judicious use of `\let` in array would have saved a register.

```
\newdimen\extrarowheight
\newdimen\tabextrasep
\newdimen\arrayextrasep
\newdimen\smarraycolsep
\newdimen\smarrayextrasep
\newdimen\tab@width
\newdimen\col@sep
\newdimen\tab@endheight
```

Some skip registers too. Phew.

```
\newskip\tab@leftskip
\newskip\tab@rightskip
```

And some switches. The first three are for the parser.

```
\newif\iftab@firstcol
\newif\iftab@initrule
\newif\iftab@rule
\newif\iftab@vgap
```

Now assign some default values to new dimen parameters. These definitions are essentially the equivalent of an `\openup 1\jot` in array, but not in tabular. This looks nice, I think.

```
\tabextrasep\z@
\arrayextrasep\jot
\smarraycolsep\thr@@\p@
\smarrayextrasep\z@
```

Set some things up for alien table environments.

```
\let\tab@extrasep\tabextrasep
\let\tab@penalty\relax
```

2.2 Some little details

`\@maybe@unskip` This macro solves a little problem. In an alignment (and in other places) it's desirable to suppress trailing space. The usual method, to say `\unskip`, is a little hamfisted, because it removes perfectly reasonable aligning spaces like `\hfil`s. While as a package writer I can deal with this sort of thing by saying `\kern\z@` in appropriate places, it can annoy users who are trying to use `\hfill` to override alignment in funny places.

My current solution seems to be acceptable. I'll remove the natural width of the last glue item, so that it can still stretch and shrink if necessary. The implementation makes use of the fact that multiplying a ⟨*skip*⟩ by a ⟨*number*⟩ kills off the stretch. (Bug fix: don't do this when we're in vertical mode.)

```
\def\@maybe@unskip{\ifhmode\hskip\m@ne\lastskip\relax\fi}
```

`\q@delim` Finally, for the sake of niceness, here's a delimiter token I can use for various things. It's a 'quark', for what it's worth (i.e., it expands to itself) although I'm not really sure why this is a good thing. As far as I'm concerned, it's important that it has a unique meaning (i.e., that it won't be `\ifx`-equal to other things, or something undefined) and that it won't be used where I don't expect it to be used.

TEX will loop horridly if it tries to expand this, so I don't think that quarks are wonderfully clever thing to use. (Maybe it should really expand to something like '⟨*quark*⟩.', which will rapdly fill TEX's memory if it gets accidentally expanded. Still, I'll leave it as it is until such time as I understand the idea more.)

```
\def\q@delim{\q@delim}
```

2.3 Parser states

Now we start on the parser. It's really simple, deep down. We progress from state to state, extracing tokens from the preamble and building command names from them. Each command calls one of the element-building routines, which works out which state it should be in. We go through each of the states in between (see later) doing default things for the ones we missed out.

Anyway, here's some symbolic names for the states. It makes my life easier.

```
\chardef\tab@startstate 0
\chardef\tab@loopstate 1
\chardef\tab@rulestate 1
\chardef\tab@prespcstate 2
\chardef\tab@prestate 3
\chardef\tab@colstate 4
\chardef\tab@poststate 5
\chardef\tab@postspcstate 6
\chardef\tab@limitstate 7
```

2.4 Adding things to token lists

Define some macros for adding stuff to the beginning and end of token lists. This is really easy, actually. Here we go.

```
\def\tab@append#1#2{#1\expandafter{\the#1#2}}
\def\tab@prepend#1#2{%
  \toks@{#2}#1\expandafter{\the\expandafter\toks@\the#1}%
}
```

2.5 Committing a column to the preamble

Each time we pass the 'rule' state, we 'commit' the tokens we've gathered so far to the main preamble token list. This is how we do it. Note the icky use of `\expandafter`.

```
\def\tab@commit{%
```

If this isn't the first column, then we need to put in a column separator.

```
  \iftab@firstcol\else%
    \expandafter\tab@append\expandafter\tab@preamble%
      \expandafter{\tab@tabtext}%
  \fi%
```

Now we spill the token registers into the main list in a funny order (which is why we're doing it in this strange way in the first place.

```
  \toks@\expandafter{\tab@midtext}%
  \tab@preamble\expandafter{%
    \the\expandafter\tab@preamble%
```

```
    \the\expandafter\tab@pretext%
    \the\expandafter\tab@userpretext%
    \the\expandafter\toks@%
    \the\tab@posttext%
  }%
```

Now reset token lists and things for the next go round.

```
  \tab@firstcolfalse%
  \tab@pretext{}%
  \tab@userpretext{}%
  \tab@posttext{}%
}
```

2.6 Playing with parser states

`\tab@setstate` This is how we set new states. The algorithm is fairly simple, really.

while *tab_state* $\neq s$ **do**
  tab_state = *tab_state* + 1;
  if *tab_state* = *tab_limitState* **then** *tab_state* = *tab_loopState*;
  if *tab_state* = *tab_preSpcState* **then**
    if *tab_initRule* **then**
      tab_initRule = **false**;
    else
      if *tab_inMultiCol* **then moan**;
      commit;
      append(*tab_shortLine*, '`&\omit`');
    end if;
  end if;
  if *tab_state* $\neq s$ **then** *do_default*(*tab_state*);
end while;

First we decide if there's anything to do. If so, we call another macro to do it for us.

```
\def\tab@setstate#1{%
  \ifnum#1=\tab@state\else%
    \def\@tempa{\tab@setstate@i{#1}}%
    \@tempa%
  \fi%
}
```

This is where the fun is. First we bump the state by one, and loop back if we fall off the end.

```
\def\tab@setstate@i#1{%
  \global\advance\tab@state\@ne%
  \ifnum\tab@state>\tab@limitstate%
    \global\tab@state\tab@loopstate%
  \fi%
```

Now, if we've just passed the ruleoff state, we commit the current text *unless* this was the strange initial rule at the very beginning. We provide a little hook here so that `\multicolumn` can moan if you try and give more than one column there. We also add another tab/omit pair to the list we use for `\vgap`.

```
  \ifnum\tab@state=\tab@prespcstate%
    \iftab@initrule%
      \tab@initrulefalse%
    \else%
      \tab@looped%
      \tab@commit%
      \tab@append\tab@shortline{&\omit}%
    \fi%
  \fi%
```

Now we decide whether to go round again. If not, we do the default thing for this state. This is mainly here so that we can put the `\tabcolsep` or whatever in if the user didn't give an '`@`' expression.

```
  \ifnum#1=\tab@state%
    \let\@tempa\relax%
  \else%
    \csname tab@default@\number\tab@state\endcsname%
  \fi%
  \@tempa%
}
```

Now we set up the default actions for the various states.

In state 2 (pre-space) we add in the default gap if either we didn't have an '`@`' expression in the post-space state or there was an explicit intervening rule.

```
\@namedef{tab@default@2}{%
  \iftab@rule%
    \tab@append\tab@pretext{\hskip\col@sep}%
  \fi%
}
```

If the user omits the column type, we insert an 'l'-type column and moan a lot.

```
\@namedef{tab@default@4}{%
  \tab@err@misscol%
  \tab@append\tab@pretext{\tab@bgroup\relax}%
  \tab@append\tab@posttext{\relax\tab@egroup\hfil}%
  \tab@append\tab@shortline{\hfil}%
  \advance\tab@columns\@ne%
}
```

Finally we deal with the post-space state. We set a marker so that we put in the default space in the pre-space state later too.

```
\@namedef{tab@default@6}{%
  \tab@append\tab@posttext{\hskip\col@sep}%
  \tab@ruletrue%
}
```

2.7 Declaring token types

`\tab@extracol` Before we start, we need to handle `\extracolsep`. This is a right pain, because the original version of tabular worked on total expansion, which is a Bad Thing. On the other hand, turning `\extracolsep` into a `\tabskip` is also a major pain.

```
\def\tab@extracol#1#2{\tab@extracol@i#1#2\extracolsep{}\extracolsep\end}
\def\tab@extracol@i#1#2\extracolsep#3#4\extracolsep#5\end{%
  \ifx @#3@%
    \def\@tempa{#1{#2}}%
  \else%
    \def\@tempa{#1{#2\tabskip#3\relax#4}}%
  \fi%
  \@tempa%
}
```

This is where we do the work for inserting preamble elements.

\tabruletype Inserting rules is interesting, because we have to decide where to put them. If this is the funny initial rule, it goes in the pre-text list, otherwise it goes in the post-text list. We work out what to do first thing:

```
\def\tabruletype#1{\tab@extracol\tabruletype@i{#1}}%
\def\tabruletype@i#1{%
  \iftab@initrule%
    \let\tab@tok\tab@pretext%
  \else%
    \let\tab@tok\tab@posttext%
  \fi%
```

Now if we're already in the rule state, we must have just done a rule. This means we must put in the `\doublerulesep` space, both here and in the shortline list. Otherwise we just stick the rule in.

This is complicated, because `\vgap` needs to be able to remove some bits of rule. We pass each one to a macro `\tab@ckr`, together with the column number, which is carefully bumped at the right times, and this macro will vet the rules and output the appropriate ones. There's lots of extreme `\expandafter` nastiness as a result. Amazingly, this actually works.

```
  \ifnum\tab@state=\tab@rulestate%
    \tab@append\tab@tok{\hskip\doublerulesep\begingroup#1\endgroup}%
    \expandafter\tab@append\expandafter\tab@shortline\expandafter{%
      \expandafter\hskip\expandafter\doublerulesep%
      \expandafter\tab@ckr\expandafter{\the\tab@columns}%
        {\begingroup#1\endgroup}%
    }%
  \else%
    \tab@setstate\tab@rulestate%
    \tab@append\tab@tok{\begingroup#1\endgroup}%
    \expandafter\tab@append\expandafter\tab@shortline\expandafter{%
      \expandafter\tab@ckr\expandafter{\the\tab@columns}%
        {\begingroup#1\endgroup}%
    }%
  \fi%
```

Finally, we say there was a rule here, so that default space gets put in after this. Otherwise we lose lots of generality.

```
  \tab@ruletrue%
}
```

\tabspctype We need to work out which space-state we should be in. Then we just put the text in. Easy, really.

```
\def\tabspctype#1{\tab@extracol\tabspctype@i{#1}}%
\def\tabspctype@i#1{%
  \tab@rulefalse%
  \ifnum\tab@state>\tab@prespcstate%
    \tab@setstate\tab@postspcstate%
    \let\tab@tok\tab@posttext%
  \else%
    \tab@setstate\tab@prespcstate%
    \let\tab@tok\tab@pretext%
  \fi%
  \tab@append\tab@tok{\begingroup#1\endgroup}%
}
```

\tabcoltype If we're already in the column state, we bump the state and loop round again, to get all the appropriate default behaviour. We bump the column counter, and add the bits of text we were given to appropriate token lists. We also add the `\hfil` glue to the shortline list, to space out the rules properly.

```
\def\tabcoltype#1#2{%
  \ifnum\tab@state=\tab@colstate%
    \global\advance\tab@state\@ne%
  \fi%
  \advance\tab@columns\@ne%
  \tab@setstate\tab@colstate%
  \tab@append\tab@pretext{#1}%
  \tab@append\tab@posttext{#2}%
  \tab@append\tab@shortline{\hfil}%
}
```

\tabuserpretype
\tabuserposttype These are both utterly trivial.

```
\def\tabuserpretype#1{%
  \tab@setstate\tab@prestate%
  \tab@prepend\tab@userpretext{#1}%
}
```

```
\def\tabuserposttype#1{%
  \tab@setstate\tab@poststate%
  \tab@prepend\tab@posttext{#1}%
}
```

2.8 The colset stack

Let's start with something fairly easy. We'll keep a stack of column sets so that users don't get confused by package authors changing the current column set. This is fairly easy, really.

\tab@push
\tab@pop
\tab@head These are the stack management routines. The only important thing to note is that `\tab@head` must take place *only* in TEX's mouth, so we can use it in `\csname`...`\endcsname` constructions.

```
\def\tab@push#1#2{%
  \toks@{{#2}}%
```

```
  \expandafter\def\expandafter#1\expandafter{\the\expandafter\toks@#1}%
}
\def\tab@pop#1{\expandafter\def\expandafter#1\expandafter{\@gobble#1}}
\def\tab@head#1{\expandafter\tab@head@i#1\relax}
\def\tab@head@i#1#2\relax{#1}
```

\colset \colpush \colpop

Now we can define the user macros.

```
\def\tab@colstack{{tabular}}
\def\colset{\colpop\colpush}
\def\colpush{\tab@push\tab@colstack}
\def\colpop{\tab@pop\tab@colstack}
```

\tab@colset

Now we define a shortcut for reading the top item off the stack.

```
\def\tab@colset{\tab@head\tab@colstack}
```

2.9 The main parser routine

\tab@initread

This macro sets up lots of variables to their normal states prior to parsing a preamble. Some things may need changing, but not many.

```
\def\tab@initread{%
```

First, reset the parser state to the start state.

```
  \global\tab@state\tab@startstate%
```

We clear the token lists to sensible values, mostly. The midtext macro contains what to put in the very middle of each template – `\multicolumn` will insert its argument here.

```
  \tab@preamble{}%
  \tab@shortline{}%
  \def\tab@tabtext{&}%
  \def\tab@midtext{\ignorespaces####\@maybe@unskip}%
  \tab@pretext{}%
  \tab@userpretext{}%
  \tab@posttext{}%
  \let\tab@multicol\@empty%
  \def\tab@startpause{\penalty\postdisplaypenalty\medskip}%
  \def\tab@endpause{\penalty\predisplaypenalty\medskip}%
```

Finally, reset the column counter, don't raise errors when we loop, and set some parser flags to their appropriate values.

```
  \tab@columns\z@%
  \let\tab@looped\relax%
  \tab@ruletrue%
  \tab@initruletrue%
  \tab@firstcoltrue%
}
```

\tab@readpreamble

This is the main macro for preamble handling. Actually, all it does is gobble its argument's leading brace and call another macro, but it does it with style.

```
\def\tab@readpreamble#1{%
  \tab@doreadpream{#1}%
  \iftab@initrule\global\tab@state\tab@prespcstate\fi%
```

```
  \tab@setstate\tab@rulestate%
  \tab@commit%
}
```

`\tab@doreadpream` The preamble is in an argument. Previous versions used a nasty trick using `\let` and `\afterassignment`. Now we use an explicit end token, to allow dodgy column type handlers to scoop up the remaining preamble tokens and process them. Not that anyone would want to do that, oh no (see the '`[`' type in the eqnarray environment `;-`)).

```
\def\tab@doreadpream#1{\tab@mkpreamble#1\q@delim}
```

`\tab@mkpreamble` This is the main parser routine. It takes each token in turn, scrutinises it carefully, and does the appropriate thing with it.

The preamble was given as an argument to `\tab@doreadpream`, and that has helpfully stripped off the initial `{` character. We need to pick off the next token (whatever it is) so we can examine it. We'll use `\futurelet` so we can detect groups and things in funny places.

```
\def\tab@mkpreamble{\futurelet\@let@token\tab@mkpreamble@i}
```

If we find a space token, we'll go off and do something a bit special, since spaces are sort of hard to handle. Otherwise we'll do it in the old fashioned way.

```
\def\tab@mkpreamble@i{%
  \ifx\@let@token\@sptoken%
    \expandafter\tab@mkpreamble@spc%
  \else%
    \expandafter\tab@mkpreamble@ii%
  \fi%
}
```

If we find a `\@@endpreamble` token, that's it and we're finished. We just gobble it and return. Otherwise, if it's an open group character, we'll complain because someone's probably tried to put an argument in the wrong place. Finally, if none of the other things apply, we'll deal with the character below.

```
\def\tab@mkpreamble@ii{%
  \ifx\@let@token\q@delim%
    \def\@tempa{\let\@let@token}%
  \else%
    \ifcat\bgroup\noexpand\@let@token%
      \tab@err@oddgroup%
      \def\@tempa##1{\tab@mkpreamble}%
    \else%
      \let\@tempa\tab@mkpreamble@iii%
    \fi%
  \fi%
  \@tempa%
}
```

Handle a character. This involves checking to see if it's actually defined, and then doing it. Doing things this way means we won't get stranded in mid-preamble unless a package author has blown it.

```
\def\tab@mkpreamble@iii#1{%
  \@ifundefined{\tab@colset!col.\string#1}{%
```

```
    \tab@err@undef{#1}\tab@mkpreamble%
  }{%
    \@nameuse{\tab@colset!col.\string#1}%
  }%
}
```

If we get given a space character, we'll look up the command name as before. If no-one's defined the column type we'll just skip it silently, which lets users do pretty formatting if they like.

```
\@namedef{tab@mkpreamble@spc} {%
  \@ifundefined{\tab@colset!col. }{%
    \tab@mkpreamble%
  }{%
    \@nameuse{\tab@colset!col. }%
  }%
}
```

`\coldef` Here's how to define column types the nice way. Some dexterity is required to make everything work right, but it's simple really.

```
\def\coldef{\@ifnextchar[\coldef@i{\coldef@i[\tab@colset]}}
\def\coldef@i[#1]#2#3#{\coldef@ii[#1]{#2}{#3}}
\def\coldef@ii[#1]#2#3#4{%
  \expandafter\def\csname#1!col.\string#2\endcsname#3{%
    #4\tab@mkpreamble%
  }%
}
```

`\collet` We'd like to let people copy column types from other places. This is how to do it.

```
\def\collet{\@ifnextchar[\collet@i{\collet@i[\tab@colset]}}
\def\collet@i[#1]#2{%
  \@ifnextchar=%
    {\collet@ii[#1]{#2}}%
    {\collet@ii[#1]{#2}=}%
}
\def\collet@ii[#1]#2={%
  \@ifnextchar[%
    {\collet@iii[#1]{#2}}%
    {\collet@iii[#1]{#2}[\tab@colset]}%
}
\def\collet@iii[#1]#2[#3]#4{%
  \expandafter\let\csname#1!col.\string#2\expandafter\endcsname%
                  \csname#3!col.\string#4\endcsname%
}
```

`\newcolumntype` We just bundle the text off to `\newcommand` and expect it to cope. It ought to. The column type code inserts the user's tokens directly, rather than calling `\tab@doreadpream` recursively. The magic control sequence is the one looked up by the parser.

There's some additional magic here for compatiblity with the obscure way that `array` works.

```
\def\newcolumntype#1{\@ifnextchar[{\nct@i{#1}}{\nct@i#1[0]}}
\def\nct@i#1[#2]{\@ifnextchar[{\nct@ii{#1}[#2]}{\nct@iii{#1}{[#2]}}}
\def\nct@ii#1[#2][#3]{\nct@iii{#1}{[#2][#3]}}
```

```
\def\nct@iii#1#2#3{%
  \expandafter\let\csname\tab@colset!col.\string#1\endcsname\relax%
  \expandafter\newcommand\csname\tab@colset!col.\string#1\endcsname#2{%
    \tab@deepmagic{#1}%
    \tab@mkpreamble%
    #3%
  }%
}
```

Now for some hacking for compatibility with tabularx.

```
\def\newcol@#1[#2]{\nct@iii{#1}{[#2]}}
```

And now some more. This is seriously deep magic. Hence the name.

```
\def\tab@deepmagic#1{%
  \csname NC@rewrite@\string#1\endcsname\NC@find\tab@@magic@@%
}
\def\NC@find#1\tab@@magic@@{}
```

2.10 Standard column types

First, make sure we're setting up the right columns. This also sets the default for the user. Other packages must not use the `\colset` command for defining columns – they should use the stack operations defined above.

```
\colset{tabular}
```

Now do the simple alignment types. These are fairly simple. The mysterious kern in the '`l`' type is to stop the `\col@sep` glue from vanishing due to the `\unskip` inserted by the standard `\tab@midtext` if the column contains no text. (Thanks for spotting this bug go to that nice Mr Carlisle.)

```
\coldef l{\tabcoltype{\kern\z@\tab@bgroup}{\tab@egroup\hfil}}
\coldef c{\tabcoltype{\hfil\tab@bgroup}{\tab@egroup\hfil}}
\coldef r{\tabcoltype{\hfil\tab@bgroup}{\tab@egroup}}
```

Some extensions now. These are explicitly teextual or mathematical columns. Can be useful if you're providing column types for other people. I've inserted a kern here for exactly the same reason as for the '`l`' column type above.

```
\coldef T#1{\tab@aligncol{#1}{\tab@btext}{\tab@etext}}
\coldef M#1{\tab@aligncol{#1}{\tab@bmaths}{\tab@emaths}}
\def\tab@aligncol#1#2#3{%
  \if#1l\tabcoltype{\kern\z@#2}{#3\hfil}\fi%
  \if#1c\tabcoltype{\hfil#2}{#3\hfil}\fi%
  \if#1r\tabcoltype{\hfil#2}{#3}\fi%
}
```

Now for the default rules.

```
\coldef |{\tabruletype{\vrule\@width\arrayrulewidth}}
\coldef !#1{\tabruletype{#1}}
```

Deal with '`@`' expressions.

```
\coldef @#1{\tabspctype{#1}}
```

And the paragraph types. I've added things to handle footnotes here.

```
\coldef p#1{\tabcoltype%
             {\savenotes\vtop\tab@bpar{#1}}%
             {\tab@epar\spewnotes\hfil}}
\coldef m#1{\tabcoltype%
             {\savenotes$\vcenter\tab@bpar{#1}}%
             {\tab@epar$\spewnotes\hfil}}
\coldef b#1{\tabcoltype%
             {\savenotes\vbox\tab@bpar{#1}}%
             {\tab@epar\spewnotes\hfil}}
```

Phew. Only a few more left now. The user text ones.

```
\coldef >#1{\tabuserpretype{#1}}
\coldef <#1{\tabuserposttype{#1}}
```

The strange column type.

```
\coldef ##1#2{\tabcoltype{#1}{#2}}
```

And '`*`', which repeats a preamble spec. This is really easy, and not at all like the original one.

```
\coldef *#1#2{%
  \count@#1%
  \loop\ifnum\count@>0\relax%
    \tab@doreadpream{#2}%
    \advance\count@\m@ne%
  \repeat%
}
```

2.11 Paragraph handling

First of all, starting new paragraphs: the vbox token is already there, and we have the width as an argument.

`\tab@bpar` There are some gymnastics to do here to support lists which form the complete text of the parbox. One of the odd things I'll do here is to not insert a strut on the first line: instead, I'll put the text into a box register so that I can inspect it later. So that I have access to the height of the first line, I'll use a `\vtop` – I can get at the final depth by using `\prevdepth`, so this seems to be the most general solution.

```
\def\tab@bpar#1{%
  \bgroup%
  \hsize#1\relax%
  \@arrayparboxrestore%
  \setbox\z@\vtop\bgroup
  \global\@minipagetrue%
  \everypar{%
    \global\@minipagefalse%
    \everypar{}%
  }%
}
```

`\tab@epar` To end the paragraph, close the box. That sounds easy, doesn't it? I need to space out the top and bottom of the box so that it looks as if struts have been applied.

```
\def\tab@epar{%
```

Anyway, I should end the current paragraph if I'm still in horizontal mode. A simple \par will do this nicely. I'll also remove any trailing vertical glue (which may be left there by a list environment), because things will look very strange otherwise.

```
\ifhmode\@maybe@unskip\par\fi%
\unskip%
```

Now I'll look at the depth of the last box: if it's less deep than my special strut, I'll cunningly backpedal by a bit, and add a box with the appropriate depth. Since this will lie on the previous baseline, it won't alter the effective height of the box. There's a snag here. \prevdepth may be wrong for example if the last thing inserted was a rule, or the box is just empty. Check for this specially. (Thanks to Rowland for spotting this.)

```
\ifdim\prevdepth>-\@m\p@\ifdim\prevdepth<\dp\@arstrutbox%
  \kern-\prevdepth%
  \nointerlineskip%
  \vtop to\dp\@arstrutbox{}%
\fi\fi%
```

I've finished the bottom of the box now: I'll close it, and start work on the top again.

```
\egroup%
```

For top-alignment to work, the first item in the box must be another box. (This is why I couldn't just set \prevdepth at the beginning.) If the box isn't high enough, I'll add a box of the right height and then kern backwards so that the 'real' first box ends up in the right place.

```
\ifdim\ht\z@<\ht\@arstrutbox%
  \vbox to\ht\@arstrutbox{}%
  \kern-\ht\z@%
\fi%
\unvbox\z@%
\egroup%
}
```

2.12 Gentle persuasion

To persuade longtable to work, we emulate some features of the array way of doing things. It's a shame, but we have to do it, because longtable came first.

Note the horribleness with the grouping here. In order to get everything expanded at the right time, \@preamble just replaces itself with the (not expanded!) preamble string, using \the. This means that the preamble string must be visible in the group just above us. Now, longtable (and array for that matter) does \@mkpreamble immediately after opening a new group. So all we need to do is close that group, do our stuff, and reopen the group again. (Evil laughter...)

```
\def\@mkpream#1{%
  \endgroup%
  \colset{tabular}%
  \tab@initread%
  \def\tab@multicol{\@arstrut}%
  \tab@preamble{\tab@multicol}%
```

```
  \def\tab@midtext{\ignorespaces\@sharp\@sharp\@maybe@unskip}%
  \tab@readpreamble{#1}%
  \gdef\@preamble{\the\tab@preamble}%
  \let\tab@bgroup\begingroup%
  \let\tab@egroup\endgroup%
  \begingroup%
}
```

2.13 Debugging

This macro just parses a preamble and displays it on the terminal. It means I can see whether the thing's working.

```
\def\showpream#1{%
  \tab@initread%
  \tab@readpreamble{#1}%
  \showthe\tab@preamble%
  \showthe\tab@shortline%
}
```

A quick macro for showing column types.

```
\def\showcol#1{%
  \expandafter\show\csname\tab@colset!col.\string#1\endcsname%
}
```

2.14 The tabular and array environments

This is where we define the actual environments which users play with.

2.14.1 The environment routines

The real work is done in the `\@array` macro later. We just set up lots (and I mean *lots*) of parameters first, and then call `\@array`.

`\tab@array` The `\tab@array` macro does most of the common array things.

```
\def\tab@array{%
  \tab@width\z@%
  \let\tab@bgroup\tab@bmaths%
  \let\tab@egroup\tab@emaths%
  \@tabarray%
}
```

`\tab@btext` `\tab@bmaths` `\tab@etext` `\tab@emaths` These macros contain appropriate things to use when typesetting text or maths macros. They're all trivial. They're here only for later modification by funny things like the smarray environment.

```
\def\tab@btext{\begingroup}
\def\tab@bmaths{$}
\def\tab@etext{\endgroup}
\def\tab@emaths{\m@th$}
```

`array` Now for the array environment. The '$' signs act as a group, so we don't need to do extra grouping this time. Closing the environment is easy.

```
\def\array{%
```

```
  \col@sep\arraycolsep%
  \let\tab@extrasep\arrayextrasep%
  \tab@normalstrut%
  \tab@array%
}
\def\endarray{%
  \crcr%
  \egroup%
  \tab@right%
  \tab@restorehlstate%
}
```

`smarray` Now for something a little different. The smarray environment gives you an array with lots of small text.

```
\def\smarray{%
  \extrarowheight\z@%
  \col@sep\smarraycolsep%
  \let\tab@extrasep\smarrayextrasep%
  \def\tab@bmaths{$\scriptstyle}%
  \def\tab@btext{\begingroup\scriptsize}%
  \setbox\z@\hbox{\scriptsize\strut}%
  \dimen@\ht\z@\dimen\tw@\dp\z@\tab@setstrut%
  \tab@array%
}
\let\endsmarray\endarray
```

`\tabstyle` This is a little hook that document designers can use to modify the appearance of tables throughout a document. For example, I've set it to make the text size `\small` in all tables in this document. Macro writers shouldn't try to use it as a hook for their own evilness, though. I've used `\providecommand` to avoid nobbling an existing definition.

```
\providecommand\tabstyle{}
```

`\@tabular` The two tabular environments share lots of common code, so we separate that out. (This needs to be done better.) All we really do here is set up the `\tab@bgroup` and `\tab@egroup` to localise things properly, and then go.

```
\def\@tabular#1{%
  \tabstyle%
  \tab@width#1%
  \let\tab@bgroup\tab@btext%
  \let\tab@egroup\tab@etext%
  \col@sep\tabcolsep%
  \let\tab@extrasep\tabextrasep%
  \tab@normalstrut%
  \@tabarray%
}
```

`tabular` `tabular*` These environments just call a macro which does all the common stuff.

```
\def\tabular{\@tabular\z@}
\expandafter\let\csname tabular*\endcsname\@tabular
\let\endtabular\endarray
\expandafter\let\csname endtabular*\endcsname\endarray
```

2.14.2 Setting the strut height

\tab@setstrut We use a magical strut, called \@arstrut, which keeps the table from collapsing around our heads. This is where we set it up.

It bases the array strut size on the given values of \dimen@ and \dimen\tw@, amended by various appropriate fiddle values added in by various people.

```
\def\tab@setstrut{%
  \setbox\@arstrutbox\hbox{%
    \vrule%
      \@height\arraystretch\dimen@%
      \@depth\arraystretch\dimen\tw@%
      \@width\z@%
  }%
}
```

\tab@normalstrut This sets the strut the normal way, from the size of \strutbox.

```
\def\tab@normalstrut{%
  \dimen@\ht\strutbox\advance\dimen@\extrarowheight%
  \dimen\tw@\dp\strutbox%
  \tab@setstrut%
}
```

2.14.3 Setting up the alignment

The following bits are mainly for other packages to hook themselves onto.

```
\let\@arrayleft\relax%
\let\@arrayright\relax%
```

```
\def\@tabarray{%
  \let\@arrayleft\relax%
  \let\@arrayright\relax%
  \@ifnextchar[\@array{\@array[c]}%
}
```

\@array The \@array macro does most of the real work for the environments. The first job is to set up the row strut, which keeps the table rows at the right height. We just take the normal strut box, and extend its height by the \extrarowheight length parameter.

```
\def\@array[#1]#2{%
```

Sort out the hline state variable. We'll store the old value in a control sequence to avoid wasting any more count registers.

```
  \edef\tab@restorehlstate{%
    \global\tab@endheight\the\tab@endheight%
    \gdef\noexpand\tab@hlstate{\tab@hlstate}%
  }%
  \def\tab@hlstate{n}%
```

Now we read the preamble. All the clever things we've already done are terribly useful here.

The \tab@setcr sets up \\ to be a newline even if users have changed it using something like \raggedright.

```
  \colset{tabular}%
```

```
  \tab@initread%
  \def\tab@midtext{\tab@setcr\ignorespaces####\@maybe@unskip}%
  \def\tab@multicol{\@arstrut\tab@startrow}%
  \tab@preamble{\tab@multicol\tabskip\z@skip}%
  \tab@readpreamble{#2}%
```

Set up the default tabskip glue. This is easy: there isn't any.

```
  \tab@leftskip\z@skip%
  \tab@rightskip\z@skip%
```

Now set up the positioning of the table. This is put into a separate macro because it's rather complicated.

```
  \tab@setposn{#1}%
```

Now work out how to start the alignment.

```
  \ifdim\tab@width=\z@%
    \def\tab@halign{}%
  \else%
    \def\tab@halign{to\tab@width}%
  \fi%
```

Finally, do all the normal things we need to do before an alignment. Note that we define `\tabularnewline` first, then set `\\` from that (using `\tab@setcr`). Since `\\` is reset in the `\tab@midtext` of every table cell, it becomes secondary to `\tabularnewline`. Doing things this way avoids the problems with declarations like `\raggedright` which redefine `\\` in their own (usually rather strange) way, so you don't need to mess about with things like the `\PreserveBackslash` command given in the *LaTeX Companion*.

```
  \lineskip\z@\baselineskip\z@%
  \m@th%
  \def\tabularnewline{\tab@arraycr\tab@penalty}%
  \tab@setcr%
  \let\par\@empty%
  \everycr{}\tabskip\tab@leftskip%
  \tab@left\halign\tab@halign\expandafter\bgroup%
    \the\tab@preamble\tabskip\tab@rightskip\cr%
}
```

You've no doubt noticed the `\tab@left` and `\tab@right` macros above. These are set up here and elsewhere to allow other things to gain control at various points of the table (they include and take the place of the `\@arrayleft` and `\@arrayright` hooks in array, put in for delarray's use.

2.14.4 Positioning the table

`\tab@setposn` This macro sets everything up for the table's positioning. It's rather long, but not all that complicated. Honest.

First, we set up some defaults (for centring). If anything goes wrong, we just do the centring things.

```
\def\tab@setposn#1{%
  \def\tab@left{%
    \savenotes%
    \leavevmode\hbox\bgroup$\@arrayleft\vcenter\bgroup%
```

```
  }%
  \def\tab@right{%
    \egroup%
    \m@th\@arrayright$\egroup%
    \spewnotes%
  }%
  \global\tab@endheight\z@%
```

For the standard positioning things, we just do appropriate boxing things. Note that the dollar signs are important, since delarray might want to put its delimiters in here.

The `\if@tempswa` switch it used to decide if we're doing an unboxed tabular. We'll set it if we find an unbox-type position code, and then check that everything's OK for this.

```
  \@tempswafalse%
  \let\tab@penalty\relax%
  \if#1t%
    \def\tab@left{%
      \savenotes%
      \leavevmode\setbox\z@\hbox\bgroup$\@arrayleft\vtop\bgroup%
    }%
    \def\tab@right{%
      \egroup%
      \m@th\@arrayright$\egroup%
      \tab@raisebase%
      \spewnotes%
    }%
    \gdef\tab@hlstate{t}%
    \global\tab@endheight\ht\@arstrutbox%
  \else\if#1b%
    \def\tab@left{%
      \savenotes%
      \leavevmode\setbox\z@\hbox\bgroup$\@arrayleft\vbox\bgroup%
    }%
    \def\tab@right{%
      \egroup%
      \m@th\@arrayright$\egroup%
      \tab@lowerbase%
      \spewnotes%
    }%
    \gdef\tab@hlstate{b}%
  \else%
    \if#1L\@tempswatrue\fi%
    \if#1C\@tempswatrue\fi%
    \if#1R\@tempswatrue\fi%
  \fi\fi%
```

Now for some tests to make sure we're allowed to do the unboxing. We text for `\@arrayleft` being defined, because people trying to hook us won't understand unboxed tabulars.

```
  \if@tempswa\ifhmode%
    \ifinner\tab@err@unbrh\@tempswafalse\else\par\fi%
  \fi\fi%
  \if@tempswa\ifmmode\tab@err@unbmm\@tempswafalse\fi\fi%
```

```
\if@tempswa\ifx\@arrayleft\relax\else%
  \tab@err@unbext\@tempswafalse%
\fi\fi%
```

Finally, if we're still doing an unboxed alignment, we need to sort out the spacing. We know that no-one's tried to hook on to the environment, so we clear `\tab@left` and `\tab@right`.

```
\if@tempswa%
  \def\tab@left{\vskip\parskip\medskip}%
  \def\tab@right{\par\@endpetrue\global\@ignoretrue}%
```

Now we need to sort out the alignment. The only way we can do this is by playing with tabskip glue. There are two possiblities:

- If this is a straight tabular or an array, we just use infinite glue. This is reasonable, I think.
- If we have a width for the table, we calculate the fixed values of glue on either side. This is fairly easy, and forces the table to the required width.

First, set up the left and right glues to represent the prevailing margins set up by list environments. I think this is the right thing to do.

```
  \tab@leftskip\@totalleftmargin%
  \tab@rightskip\hsize%
  \advance\tab@rightskip-\linewidth%
  \advance\tab@rightskip-\@totalleftmargin%
```

First of all, deal with the simple case. I'm using 10000 fill glue here, in an attempt to suppress `\extracolsep` glue from making the table the wrong width. It can always use filll glue if it really needs to, though.

```
  \ifdim\tab@width=\z@%
    \if#1L\else\advance\tab@leftskip\z@\@plus10000fill\fi%
    \if#1R\else\advance\tab@rightskip\z@\@plus10000fill\fi%
```

Now for the fun bit. This isn't too hard really. The extra space I must add around the table adds up to `\linewidth` − `\tab@width`. I just need to add this onto the appropriate sides of the table.

```
  \else%
    \dimen@\linewidth%
    \advance\dimen@-\tab@width%
    \if#1L\advance\tab@rightskip\dimen@\fi%
    \if#1R\advance\tab@leftskip\dimen@\fi%
    \if#1C%
      \advance\tab@leftskip.5\dimen@%
      \advance\tab@rightskip.5\dimen@%
    \fi%
  \fi%
```

Don't allow page breaks. David Carlisle's wonderful longtable package does page breaks far better than I could possibly do here, and we're compatible with it (wahey!).

```
  \def\tab@penalty{\penalty\@M}%
```

Finally, set the new width of the table, and leave.

```
    \tab@width\hsize%
  \fi%
}
```

2.14.5 Handling tops and bottoms

This is how the tops and bottoms of tables are made to line up with the text on the same line, in the presence of arbitrary rules and space. The old method, based on the way the array package worked, wasn't terribly good. This new version copes much better with almost anything that gets thrown at it.

I'll keep a state in a macro (`\tab@hlstate`), which tells me what I'm meant to be doing. The possible values are '`n`', which means I don't have to do anything, '`t`', which means that I'm meant to be handling top-aligned tables, and '`b`', which means that I'm meant to be lining up the bottom. There are several other 'substates' which have various magic meanings.

```
\def\tab@hlstate{n}
```

When all's said and done, I extract the box containing the table, and play with the height and depth to try and make it correct.

`\tab@addruleheight` This macro is called by 'inter-row' things to add their height to our dimen register.

Only do this if the state indicates that it's sensible.

```
\def\tab@addruleheight#1{%
  \if\tab@hlstate n\else%
    \global\advance\tab@endheight#1\relax%
  \fi%
}
```

`\tab@startrow` This is called at the start of a row, from within the array preamble. Currently, this assumes that the rows aren't bigger than their struts: this is reasonable, although slightly limiting, and it could be done better if I was willing to rip the alignment apart and put it back together again.

```
\def\tab@startrow{%
  \if\tab@hlstate t%
    \gdef\tab@hlstate{n}%
  \else\if\tab@hlstate b%
    \global\tab@endheight\dp\@arstrutbox%
  \fi\fi%
}
```

`\tab@raisebase` This macro is called at the end of it all, to set the height and depth of the box correctly. It sets the height to `\tab@endheight`, and the depth to everything else. The box is in `\box` 0 currently.

```
\def\tab@raisebase{%
  \global\advance\tab@endheight-\ht\z@%
  \raise\tab@endheight\box\z@%
}
```

`\tab@lowerbase` And, for symmetry's sake, here's how to set the bottom properly instead.

```
\def\tab@lowerbase{%
```

```
  \global\advance\tab@endheight-\dp\z@%
  \lower\tab@endheight\box\z@%
}
```

2.15 Breaking tables into bits

Unboxed tables have a wonderful advantage over boxed ones: you can stop halfway through and do something else for a bit. Here's how:

`\tabpause` I'd like to avoid forbidding catcode changes here. I'll use `\doafter` now I've got it, to ensure that colour handling and things occur *inside* the `\noalign` (otherwise they'll mess up the alignment very seriously).

We have to be careful here to ensure that everything works correctly within lists. (The amsmath package had this problem in its `\intertext` macro, so I'm not alone here.)

```
\def\tabpause#{%
  \noalign{\ifnum0=`}\fi%
  \@parboxrestore%
  \tab@startpause%
  \vskip-\parskip%
  \parshape\@ne\@totalleftmargin\linewidth%
  \noindent%
  \doafter\tabpause@i%
}
\def\tabpause@i{%
  \nobreak%
  \tab@endpause%
  \ifnum0=`{\fi}%
}
```

2.16 The wonderful world of `\multicolumn`

`\multicolumn` This is actually fantasitcally easy. Watch and learn. Make sure you notice the `\long`s here: remember that some table cells can contain paragraphs, so it seems sensible to allow `\par` into the argument. (As far as I know, most other `\multicolumn` commands don't do this, which seems a little silly. Then again, I forgot to do it the first time around.)

```
\long\def\multicolumn#1#2#3{%
  \multispan{#1}%
  \begingroup%
    \tab@multicol%
    \tab@initread%
    \tab@preamble{}%
    \long\def\tab@midtext{#3}%
    \let\tab@looped\tab@err@multi%
    \tab@readpreamble{#2}%
    \the\tab@preamble%
  \endgroup%
  \ignorespaces%
}
```

2.17 Interlude: range lists

For processing arguments to `\vgap` and `\cline`, we need to be able to do things with lists of column ranges. To save space, and to make my fingers do less typing, here's some routines which do range handling.

`\ranges` Given a macro name and a comma separated list of ranges and simple numbers, this macro will call the macro giving it each range in the list in turn. Single numbers n will be turned into ranges n–n.

The first job is to read the macro to do (which may already have some arguments attached to it). We'll also start a group to make sure that our changes to temp registers don't affect anyone else.

There's a space before the delimiting `\q@delim` to stop numbers being parsed to far and expanding our quark (which will stop TEX dead in its tracks). Since we use `\@ifnextchar` to look ahead, spaces in range lists are perfectly all right.

```
\def\ranges#1#2{%
  \gdef\ranges@temp{#1}%
  \begingroup%
  \ranges@i#2 \q@delim%
}
```

We're at the beginning of the list. We expect either the closing marker (if this is an empty list) or a number, which we can scoop up into a scratch register.

```
\def\ranges@i{%
  \@ifnextchar\q@delim\ranges@done{\afterassignment\ranges@ii\count@}%
}
```

We've read the first number in the range. If there's another number, we'll expect a '`-`' sign to be next. If there is no '`-`', call the user's code with the number duplicated and then do the rest of the list.

```
\def\ranges@ii{%
  \@ifnextchar-\ranges@iii{\ranges@do\count@\count@\ranges@v}%
}
```

Now we strip the '`-`' off and read the other number into a temporary register.

```
\def\ranges@iii-{\afterassignment\ranges@iv\@tempcnta}
```

We have both ends of the range now, so call the user's code, passing it both ends of the range.

```
\def\ranges@iv{\ranges@do\count@\@tempcnta\ranges@v}
```

We've finished doing an item now. If we have a '`,`' next, then start over with the next item. Otherwise, if we're at the end of the list, we can end happily. Finally, if we're totally confused, raise an error.

```
\def\ranges@v{%
  \@ifnextchar,%
    \ranges@vi%
    {%
      \@ifnextchar\q@delim%
        \ranges@done%
        {\tab@err@range\ranges@vi,}%
    }%
}
```

We had a comma, so gobble it, read the next number, and go round again.

```
\def\ranges@vi,{\afterassignment\ranges@ii\count@}
```

Here's how we call the user's code, now. We close the group, so that the user's code doesn't have to do global things to remember its results, and we expand the two range ends from their count registers. We also ensure that the range is the right way round.

```
\def\ranges@do#1#2{%
  \ifnum#1>#2\else%
    \expandafter\endgroup%
    \expandafter\ranges@temp%
    \expandafter{%
    \the\expandafter#1%
    \expandafter}%
    \expandafter{%
    \the#2%
    }%
    \begingroup%
  \fi%
}
```

And finishing the scan is really easy. We close the group after gobbling the close token.

```
\def\ranges@done\q@delim{\endgroup}
```

`\ifinrange` Something a little more useful, now. `\ifinrange` takes four arguments: a number, a range list (as above), and two token lists which I'll call *then* and *else*. If the number is in the list, I'll do *then*, otherwise I'll do *else*.

```
\def\ifinrange#1#2{%
  \@tempswafalse%
  \count@#1%
  \ranges\ifinrange@i{#2}%
  \if@tempswa%
    \expandafter\@firstoftwo%
  \else%
    \expandafter\@secondoftwo%
  \fi%
}
\def\ifinrange@i#1#2{%
  \ifnum\count@<#1 \else\ifnum\count@>#2 \else\@tempswatrue\fi\fi%
}
```

2.18 Horizontal rules OK

This is where all the gubbins for `\vgap` and friends is kept, lest it contaminate fairly clean bits of code found elsewhere.

2.18.1 Drawing horizontal rules

`\hline` Note the funny use of `\noalign` to allow TEX stomach ops like `\futurelet` without starting a new table row. This lets us see if there's another `\hline` coming up, so we can see if we need to insert extra vertical space.

```
\def\hline{%
  \tab@dohline%
  \noalign{\ifnum0=`}\fi%
  \tab@penalty%
  \futurelet\@let@token\hline@i%
}
```

We check here for another `\hline` command, and insert glue if there is. This looks terrible, though, and `\hlx{hvh}` is much nicer. Still...

```
\def\hline@i{%
  \ifx\@let@token\hline%
    \vskip\doublerulesep%
    \tab@addruleheight\doublerulesep%
  \fi%
  \ifnum0=`{\fi}%
}
```

`\tab@dohline` This is where hlines actually get drawn. Drawing lines is more awkward than it used to be, particularly in unboxed tables. It used to be a case simply of saying `\noalign{\hrule}`. However, since unboxed tables are actually much wider than they look, this would make the rules stretch right across the page and look generally horrible.

The solution is simple: we basically do a dirty big `\cline`.

```
\def\tab@dohline{%
  \multispan{\tab@columns}%
  \leaders\hrule\@height\arrayrulewidth\hfil%
  \tab@addruleheight\arrayrulewidth%
  \cr%
}
```

2.18.2 Vertical rules

I couldn't fit these in anywhere else, so they'll have to go here. I'll provide a new optional argument which specifies the width of the rule; this gets rid of the problem described in the *Companion*, where to get an unusually wide vertical rule, you have to play with things like `\vrule width` ⟨*dimen*⟩ which really isn't too nice.

`\vline` The new `\vline` has an optional argument which gives the width of the rule. The `\relax` stops TEX trying to parse a ⟨*rule-specification*⟩ for too long, in case someone says something like '`\vline depthcharges`' or something equally unlikely.

```
\renewcommand\vline[1][\arrayrulewidth]{\vrule\@width#1\relax}
```

2.18.3 Drawing bits of lines

Just for a bit of fun, here's an extended version of `\cline` which takes a list of columns to draw lines under, rather than just a single range.

`\cline` Not a single line of code written yet, and we already have a dilemma on our hands. Multiple consecutive `\cline` commands are meant to draw on the same vertical bit of table. But horizontal lines are meant to have thickness now. Oh, well [sigh], we'll skip back on it after all.

Now the problem remains how best to do the job. The way I see it, there are three possibilities:

- We can start a table row, and then for each column of the table (as recorded in `\tab@columns`) we look to see if that column is listed in the range list and if so draw the rule. This requires lots of scanning of the range list.
- We can take each range in the list, and draw rules appropriately, just like the old `\cline` used to do, and starting a new table row for each.
- We can start a table row, and then for each range remember where we stopped drawing the last row, move to the start of the new one, and draw it. If we start moving backwards, we close the current row and open a new one.

The last option looks the most efficient, and the most difficult. This is therefore what I shall do `;-)`.

The first thing to do is to add in a little negative space, and start a table row (omitting the first item). Then scan the range list, and finally close the table row and add some negative space again.

We need a global count register to keep track of where we are. Mixing local and global assignments causes all sorts of tragedy, so I shall hijack `\tab@state`.

```
\def\cline#1{%
  \noalign{\kern-.5\arrayrulewidth\tab@penalty}%
  \omit%
  \global\tab@state\@ne%
  \ranges\cline@i{#1}%
  \cr%
  \noalign{\kern-.5\arrayrulewidth\tab@penalty}%
}
```

Now for the tricky bit. When we're given a range, we look to see if the first number is less than `\tab@state`. If so, we quickly close the current row, kern backwards and start again with an `\omit` and reset `\tab@state` to 1, and try again.

```
\def\cline@i#1#2{%
  \ifnum#1<\tab@state\relax%
    \tab@@cr%
    \noalign{\kern-\arrayrulewidth\tab@penalty}%
    \omit%
    \global\tab@state\@ne%
  \fi%
```

We are now either at or in front of the column position required. If we're too far back, we must `\hfil&\omit` our way over to the correct column.

```
  \@whilenum\tab@state<#1\do{%
    \hfil\tab@@tab@omit%
    \global\advance\tab@state\@ne%
  }%
```

We've found the start correctly. We must deal with a tiny problem now: if this is not the first table cell, the left hand vertical rule is in the column to the left, so our horizontal rule won't match up properly. So we skip back by a bit to compensate. If there isn't actually a vertical rule to line up with, no-one will notice, because the rules are so thin. This adds a little touch of quality to the whole thing, which is after all the point of this whole exercise.

```
  \ifnum\tab@state>\@ne%
    \kern-\arrayrulewidth%
  \fi%
```

Now we must stretch this table cell to the correct width.

```
  \@whilenum\tab@state<#2\do{%
    \tab@@span@omit%
    \global\advance\tab@state\@ne%
  }%
```

We're ready. Draw the rule. Note that this is `\hfill` glue, just in case we start putting in `\hfil` glue when we step onto the next cell.

```
  \leaders\hrule\@height\arrayrulewidth\hfill%
}
```

Some alignment primitives are hidden inside macros so they don't get seen at the wrong time. This is what they look like:

```
\def\tab@@cr{\cr}
\def\tab@@tab@omit{&\omit}
\def\tab@@span@omit{\span\omit}
```

2.18.4 Drawing short table rows

Before I start on a description of more code, I think I'll briefly discuss my reasons for leaving the `\vgap` command in its current state. There's a reasonable case for introducing an interface between `\vgap` and `\multicolumn`, to avoid all the tedious messing about with column ranges. There are good reasons why I'm not going to do this:

- It's very difficult to do: it requires either postprocessing of the table or delaying processing of each row until I know exactly what's in it; a `\multicolumn` in a row should be able to affect a `\vgap` before the row, which gets very nasty. This package is probably far too large already, and adding more complexity and running the risk of exhausting TEX's frustratingly finite capacity for the sake of relieving the user of a fairly trivial job doesn't seem worthwhile.
- Perhaps more importantly, there are perfectly valid occasions when it's useful to have the current vgap behaviour. For example, the `MIX` word layout diagrams found in *The Art of Computer Programming* use the little 'stub lines' to show where data items cross byte boundaries:

empty	−	1	0	0	0	0
occupied	+	LINK		KEY		

That's my excuses out of the way; now I'll press on with the actual programming.

`\tab@checkrule` We have a range list in `\tab@xcols` and a number as an argument. If we find the number in the list, wejust space out the following group, otherwise we let it be.

```
\def\tab@checkrule#1{%
```

```
  \count@#1\relax%
  \expandafter\ifinrange%
  \expandafter\count@%
  \expandafter{\tab@xcols}%
    {\tab@checkrule@i}%
    {}%
}
\def\tab@checkrule@i#1{\setbox\z@\hbox{#1}\hb@xt@\wd\z@{}}
```

\vgap We must tread carefully here. A single misplaced stomach operation can cause error messages. We therefore start with an \omit so we can search for optional arguments.

So that \hlx can get control after \vgap has finished, we provide a hook called \vgap@after which is expanded after \vgap has finished. Here we make it work like \@empty, which expands to nothing. (Note that \relax will start a new table row, so we can't use that.) There are some penalty items here to stick the \vgap row to the text row and \hline that are adjacent to it. The longtable package will split an \hline in half, so this is the correct thing to do.

```
\def\vgap{%
  \noalign{\nobreak}%
  \omit%
  \global\let\vgap@after\@empty%
  \iffalse{\fi\ifnum0=`}\fi%
  \@ifnextchar[\vgap@i\vgap@simple%
}
```

We set up two different sorts of \vgap – a simple one which allows all rules to be passed through, and a specific one which carefully vets each one (and is therefore slower). We decide which to so based on the presence of an optional argument.

The optional argument handler just passes its argument to an interface routine which is used by \hlx.

```
\def\vgap@i[#1]{\vgap@spec{#1}}
```

Now we handle specified columns. Since we're in an omitted table cell, we must set things up globally. Assign the column spec to a macro, and set up vetting by the routine above. Then just go and do the job.

```
\def\vgap@spec#1#2{%
  \gdef\tab@xcols{#1}%
  \global\let\tab@ckr\tab@checkrule%
  \vgap@do{#2}%
}
```

Handle all columns. Just gobble the column number for each rule, and let the drawing pass unharmed. Easy.

```
\def\vgap@simple#1{%
  \global\let\tab@ckr\@gobble%
  \vgap@do{#1}%
}
```

This is where stuff actually gets done. We set the \vgap flag on while we do the short row. Then just expand the token list we built while scanning the preamble.

Note that the flag is cleared at the end of the last column, to allow other funny things like `\noalign` and `\omit` before a new row is started.

```
\def\vgap@do#1{%
  \ifnum0=`{}\fi%
  \global\tab@vgaptrue%
  \the\tab@shortline%
    \vrule\@height#1\@width\z@%
    \global\tab@vgapfalse
    \tab@addruleheight{#1}%
    \cr%
  \noalign{\nobreak}%
  \vgap@after%
}
```

2.18.5 Prettifying syntax

`\hlx` This is like a poor cousin to the preamble parser. The whole loop is carefully written to take place *only* in TEX's mouth, so the alignment handling bits half way down the gullet don't see any of this.

First, pass the string to another routine.

```
\def\hlx#1{\hlx@loop#1\q@delim}
```

Now peel off a token, and dispatch using `\csname`. We handle undefinedness of the command in a fairly messy way, although it probably works. Maybe.

```
\def\hlx@loop#1{%
  \ifx#1\q@delim\else%
    \@ifundefined{hlx@cmd@\string#1}{%
      \expandafter\hlx@loop%
    }{%
      \csname hlx@cmd@\string#1\expandafter\endcsname%
    }%
  \fi%
}
```

`\hlxdef` New `\hlx` commands can be defined using `\hlxdef`. This is a simple abbreviation.

```
\def\hlxdef#1{\@namedef{hlx@cmd@#1}}
```

`\hlx h` Handle an '`h`' character. Just do an `\hline` and return to the loop. We look ahead to see if there's another '`h`' coming up, and if so insert two `\hline` commands. This strange (and inefficient) behaviour keeps packages which redefine `\hline` happy.

```
\hlxdef h#1{%
  \noalign{%
    \ifx#1h%
      \def\@tempa{\hline\hline\hlx@loop}%
    \else%
      \def\@tempa{\hline\hlx@loop#1}%
    \fi%
    \expandafter
  }%
  \@tempa%
}
```

`\hlx b` The ‘b’ character does a nifty backspace, for longtable’s benefit.

```
\hlxdef b{\noalign{\kern-\arrayrulewidth}\hlx@loop}
```

`\hlx /` The ‘/’ character allows a page break at the current position.

```
\hlxdef /{%
  \noalign{\ifnum0=‘}\fi%
  \@ifnextchar[\hlx@cmd@break@i{\hlx@cmd@break@i[0]}%
}
\def\hlx@cmd@break@i[#1]{\ifnum0=‘{\fi}\pagebreak[0]\hlx@loop}
```

`\hlx v` Handle a ‘v’ character. This is rather like the `\vgap` code above, although there are syntactic differences.

```
\hlxdef v{%
  \noalign{\nobreak}%
  \omit%
  \iffalse{\fi\ifnum0=‘}\fi%
  \global\let\vgap@after\hlx@loop%
  \@ifnextchar[\hlx@vgap@i{\hlx@vgap@ii\vgap@simple}%
}
\def\hlx@vgap@i[#1]{%
  \ifx!#1!%
    \def\@tempa{\hlx@vgap@ii\vgap@simple}%
  \else%
    \def\@tempa{\hlx@vgap@ii{\vgap@spec{#1}}}%
  \fi%
  \@tempa%
}
\def\hlx@vgap@ii#1{%
  \@ifnextchar[{\hlx@vgap@iii{#1}}{\hlx@vgap@iii{#1}[\doublerulesep]}%
}
\def\hlx@vgap@iii#1[#2]{#1{#2}}
```

`\hlx s` Allow the user to leave a small gap using the ‘s’ command.

```
\hlxdef s{%
  \noalign{\ifnum0=‘}\fi%
  \nobreak%
  \@ifnextchar[\hlx@space@i{\hlx@space@i[\doublerulesep]}%
}
\def\hlx@space@i[#1]{%
  \vskip#1%
  \tab@addruleheight{#1}%
  \ifnum0=‘{\fi}%
  \hlx@loop%
}
```

`\hlx c` We might as well allow a ‘c’ command to do a `\cline`.

```
\hlxdef c#1{\cline{#1}\hlx@loop}
```

`\hlx .` The ‘.’ character forces a start of the new column. There’s a little problem here. Since the ‘.’ character starts the next column, we need to gobble any spaces following the `\hlx` command before the cell contents actually starts. Unfortunately, `\ignorespaces` will start the column for us, so we can’t put it in always. We’ll

handle it here, then. We'll take the rest of the 'preamble' string, and warn if it's not empty. Then we'll `\ignorespaces` – this will start the column for us, so we don't need to `\relax` any more.

```
\hlxdef .#1\q@delim{%
  \ifx @#1@\else%
    \PackageWarning{mdwtab}{%
      Ignoring \protect\hlx\space command characters following a
      `.'\MessageBreak command%
    }%
  \fi%
  \ignorespaces%
}
```

2.19 Starting new table rows

We take a break from careful mouthery at last, and start playing with newlines. The standard one allows pagebreaks in unboxed tables, which isn't really too desirable.

Anyway, we'll try to make this macro rather more reusable than the standard one. Here goes.

`\@arraycr` We pass lots of information to a main parser macro, and expect it to cope.

```
\def\@arraycr{\tab@arraycr{}}
\def\tab@arraycr#1{\tab@cr{\tab@tabcr{#1}}{}{}}
```

Now to actually do the work. `\tab@cr` passes us the skip size, and the appropriate one of the two arguments given above (both of which are empty) depending on the presence of the $*$.

```
\def\tab@tabcr#1#2{%
```

If the total height I need to add between rows (from the optional argument and the 'extrasep' parameter) is greater than zero, I'll handle this by extending the strut slightly. I'm not actually sure whether this is the right thing to do, to be honest, although it's easier than trying to to an automatic `\vgap`, because I need to know which columns to skip. If the space is less than zero, I'll just insert the vertical space with in a `\noalign`.

First, to calculate how much space needs adding.

```
  \dimen@#2%
  \advance\dimen@\tab@extrasep%
```

If the height is greater than zero, I need to play with the strut. I must bear in mind that the current table cell (which I'm still in, remember) may be in vertical mode, and I may or may not be in a paragraph.

If I am in vertical mode, I'll backpedal to the previous box and put the strut in an hbox superimposed on the previous baseline. Otherwise, I can just put the strut at the end of the text. (This works in either LR or paragraph mode as long as I'm not between paragraphs.) Again, Rowland's empty cell bug strikes. (See `\tab@epar` for details.)

```
  \ifdim\dimen@>\z@%
    \ifvmode%
      \unskip\ifdim\prevdepth>-\@m\p@\kern-\prevdepth\fi%
```

```
      \nointerlineskip\expandafter\hbox%
    \else%
      \@maybe@unskip\expandafter\@firstofone%
    \fi%
    {\advance\dimen@\dp\@arstrutbox\vrule\@depth\dimen@\@width\z@}%
  \fi%
```

This table cell works as a group (which is annoying here). I'll copy the interrow gap into a global register so that I can use it in the `\noalign`.

```
  \global\dimen\@ne\dimen@%
  \cr%
  \noalign{%
    #1%
    \ifdim\dimen\@ne<\z@\vskip\dimen\@ne\relax\fi%
  }%
  \@gobble%
}
```

`\tab@setcr` To set the `\\` command correctly in each table cell, we make it a part of the preamble (in `\tab@midtext`) to call this routine. It's easy – just saves the preamble from being huge.

```
\def\tab@setcr{\let\\\tabularnewline}
```

`\tab@cr` Now we do the parsing work. This is fun. Note the revenge of the funny braces here. Nothing to worry about, honest. The tricky bit is to keep track of which arguments are which. (Thanks to David Carlisle for pointing out that I'd missed out the `\relax` here.)

```
\def\tab@cr#1#2#3{%
  \relax%
  \iffalse{\fi\ifnum0=`}\fi%
  \@ifstar{\tab@cr@i{#1}{#3}}{\tab@cr@i{#1}{#2}}%
}
\def\tab@cr@i#1#2{%
  \@ifnextchar[{\tab@cr@ii{#1}{#2}}{\tab@cr@ii{#1}{#2}[\z@]}%
}
\def\tab@cr@ii#1#2[#3]{%
  \ifnum0=`{}\fi%
  #1{#3}{#2}%
}
```

2.20 Gratuitous grotesquery

So far we've had an easy-ish ride (or should that be *queasy*?). Now for something unexplainably evil. We convince LaTeX that it's loaded the array package, so that packages which need it think they've got it.

The bogus date is the same as the date for the array package I've got here – this will raise a warning if Frank updates his package which should filter back to me telling me that there's something I need to know about.

The messing with `\xdef` and the funny parsing ought to insert the current mdwtab version and date into the fake array version string, giving a visible clue to the user that this isn't the real array package.

```
\begingroup
```

```
\catcode`.=11
\def\@tempa#1 #2 #3\@@{#1 #2}
\xdef\ver@array.sty
  {1995/11/19 [mdwtab.sty \expandafter\@tempa\ver@mdwtab.sty\@@]}
\endgroup
```

2.21 Error messages

I've put all the error messages together, where I can find them, translate them or whatever.

First, some token-space saving (which also saves my fingers):

```
\def\tab@error{\PackageError{mdwtab}}
```

Now do the error messages.

```
\def\tab@err@misscol{%
  \tab@error{Missing column type}{%
    I'm lost.  I was expecting something describing^^J%
    the type of the current column, but you seem to^^J%
    have missed it out.  I've inserted a type `l'^^J%
    column here in the hope that this makes sense.%
  }%
}
\def\tab@err@oddgroup{%
  \tab@error{Misplaced group in table preamble}{%
    I've found an open brace character in your preamble^^J%
    when I was expecting a specifier character.  I'm^^J%
    going to gobble the whole group and carry on as if^^J%
    I'd never seen it.%
  }%
}
\def\tab@err@undef#1{%
  \tab@error{Unknown `\tab@colset' preamble character `\string#1'}{%
    I don't understand what you meant by typing this^^J%
    character.  Anyway, I'll ignore it this time around.^^J%
    Just don't you do it again.%
  }%
}
\def\tab@err@unbrh{%
  \tab@error{Can't use unboxed tabular in LR mode}{%
    You've asked for a tabular or array environment with^^J%
    `L', `C' or `R' as the position specifier, but you're^^J%
    in LR (restricted horizontal) mode, so it won't work.^^J%
    I'll assume you really meant `c' and soldier on.%
  }%
}
\def\tab@err@unbmm{%
  \tab@error{Can't use unboxed tabular in maths mode}{%
    You've asked for a tabular or array environment with^^J%
    `L', `C' or `R' as the position specifier, but you're^^J%
    in maths mode, so it won't work.  I'll pretend that^^J%
    you really typed `c', and that this is all a bad dream.%
  }%
```

```
}
\def\tab@err@unbext{%
  \tab@error{Can't extend unboxed tabulars}{%
    You're trying to use kludgy extensions (e.g.,^^J%
    'delarray') on an array or tabular with 'L', 'C'^^J%
    or 'R' as the position specifier.  I'll assume you^^J%
    subconsciously wanted a 'c' type all along.%
  }%
}
\def\tab@err@multi{%
  \tab@error{More than one column in a \protect\multicolumn}{%
    You've put more than one column into a \string\multicolumn^^J%
    descriptor.  It won't work.  I have no idea what^^J%
    will happen, although it won't be pleasant.  Hold^^J%
    on tight now...%
  }%
}
\def\tab@err@range{%
  \tab@error{Expected ',' or '<end>' in range list}{%
    I was expecting either the end of the range list,^^J%
    or a comma, followed by another range.  I've^^J%
    inserted a comma to try and get me back on track.^^J%
    Good luck.%
  }%
}
```

That's it. No more. Move along please.

⟨/mdwtab⟩

3 Implementation of mathenv

This is in a separate package, mainly to avoid wasting people's memory.

⟨∗mathenv⟩

3.1 Options handling

We need to be able to cope with fleqn and leqno options. This will adjust our magic modified eqnarray environment appropriately.

```
\newif\if@fleqn
\newif\if@leqno
\DeclareOption{fleqn}{\@fleqntrue}
\DeclareOption{leqno}{\@leqnotrue}
\ProcessOptions
```

We use the mdwtab package for all its nice table handling things. (Oh, and to inflict it on users who want to do nice equations and don't care about our tables.)

```
\RequirePackage{mdwtab}
```

3.2 Some useful registers

The old LaTeX version puts the equation numbers in by keeping a count of where it is in the alignment. Since I don't know how may columns there are going to be, I'll just use a switch in the preamble to tell me to stop tabbing.

```
\newif\if@eqalast
```

Now define some useful length parameters. First allocate them:

```
\newskip\eqaopenskip
\newskip\eqacloseskip
\newskip\eqacolskip
\newskip\eqainskip
\newskip\splitleft
\newskip\splitright
```

Now assign some default values. Users can play with these if they really want although I can't see the point myself.

```
\AtBeginDocument{%
  \eqacloseskip\@centering%
  \eqacolskip1.5em\@plus\@m\p@
  \eqainskip\z@%
  \if@fleqn%
    \eqaopenskip\mathindent%
    \splitleft\mathindent\relax%
    \splitright\mathindent\@minus\mathindent\relax%
  \else%
    \eqaopenskip\@centering%
    \splitleft2.5em\@minus2.5em%
    \splitright\splitleft%
  \fi%
  \relax%
}
```

3.3 A little display handling

I'm probably going a little far here, and invading territory already claimed by the amsmath stuff (and done a good deal better than I can be bothered to do), but just for completeness, this is how we handle attempts to put displays inside other displays without screwing up the spacing.

`\dsp@startouter` This is how we start an outermost display. It's fairly easy really. We make `\dsp@start` start an inner display, and make `\dsp@end` close the outer display.

```
\def\dsp@startouter{%
  \let\dsp@end\dsp@endouter%
  $$%
}
```

`\dsp@endouter` Ending the outer display is utterly trivial.

```
\def\dsp@endouter{$$}
```

`\dsp@startinner` Starting inner displays is done in a vbox (actually I choose `\vbox` or `\vtop` depending on the setting of leqno to put the equation number the right way round).

```
\def\dsp@startinner{%
  \let\dsp@end\dsp@endinner%
  \if@fleqn\kern-\mathindent\fi%
  \if@leqno\vtop\else\vtop\fi\bgroup%
}
```

`\dsp@endinner` Ending an inner display is also really easy.

```
\def\dsp@endinner{\egroup}
```

`\dsp@start` This is what other bits of code uses to start displays. It's one of the start macros up above, and outer by default.

```
\def\dsp@start{%
  \ifmmode%
    \ifinner\mth@err@mdsp\fi%
    \expandafter\dsp@startinner%
  \else%
    \ifhmode\ifinner\mth@err@hdsp\fi\fi%
    \expandafter\dsp@startouter%
  \fi%
}
```

`\dsp@tabpause` This sets up the correct pre- and postambles for the `\tabpause` macro in maths displays. This is fairly simple stuff.

```
\def\dsp@tabpause{%
  \def\tab@startpause%
    {\penalty\postdisplaypenalty\vskip\belowdisplayskip}%
  \def\tab@endpause%
    {\penalty\predisplaypenalty\vskip\abovedisplayskip}%
}
```

3.4 The eqnarray environment

We allow the user to play with the style if this is really wanted. I dunno why, really. Maybe someone wants very small alignments.

```
\let\eqastyle\displaystyle
```

3.4.1 The main environments

`eqnarray` `eqnarray*` We define the toplevel commands here. They just add in default arguments and then call `\@eqnarray` with a preamble string. We handle equation numbers by setting up a default (`\eqa@defnumber`) which is put into the final column. At the beginning of each row, we globally `\let \eqa@number` equal to `\eqa@defnumber`. The `\eqnumber` macro just changes `\eqa@number` as required. Since `\eqa@number` is changed globally we must save it in this environment.

First, we must sort out the optional arguments and things. This is really easy. The only difference between the starred and non-starred environments is the default definition of `\eqa@defnumber`.

```
\def\eqnarray{%
  \eqnarray@i\eqa@eqcount%
}
\@namedef{eqnarray*}{\eqnarray@i{}}
\def\eqnarray@i#1{\@ifnextchar[{\eqnarray@ii{#1}}{\eqnarray@ii{#1}[rcl]}}
```

Right. Now for the real work. The first argument is the default numbering tokens; the second is the preamble string.

```
\def\eqnarray@ii#1[#2]{%
```

Set up the equation counter and labels correctly.

\begin{rant}
The hacking with `\@currentlabel` is here because (in the author's opinion) LaTeX's `\refstepcounter` macro is broken. It's currently defined as

```
\def\refstepcounter#1{%
  \stepcounter{#1}%
  \protected@edef\@currentlabel%
    {\csname p@#1\endcsname\csname the#1\endcsname}%
}
```

which means that the current label gets 'frozen' as soon as you do the counter step. By redefining the macro as

```
\def\refstepcounter#1{%
  \stepcounter{#1}%
  \edef\@currentlabel{%
    \expandafter\noexpand\csname p@#1\endcsname%
    \expandafter\noexpand\csname the#1\endcsname%
  }%
}
```

these sorts of problems would be avoided, without any loss of functionality or compatibility that I can see.
\end{rant}

```
  \stepcounter{equation}%
  \def\@currentlabel{\p@equation\theequation}%
```

The next step is to set up the numbering. I must save the old numbering so I can restore it later (once in the alignment, I must assign these things globally).

```
  \let\eqa@oldnumber\eqa@number%
  \def\eqa@defnumber{#1}%
  \global\let\eqa@number\eqa@defnumber%
```

The `\if@eqalastfalse` switch is false everywhere except when we're in the final column.

```
  \@eqalastfalse%
```

Remove the `\mathsurround` kerning, since it will look very odd inside the display. We have our own spacing parameters for configuring these things, so `\mathsurround` is unnecessary.

```
  \m@th%
```

Time to parse the preamble string now. I must choose the correct column set, initialise the preamble parser and set up the various macros. The extra '`@{\tabskip\eqacloseskip}`' item sets up the tabskip glue to centre the alignment properly.

```
  \colset{eqnarray}%
  \tab@initread%
```

```
  \def\tab@tabtext{&\tabskip\z@skip}%
  \tab@preamble{\tabskip\z@skip}%
  \tab@readpreamble{#2@{\tabskip\eqacloseskip}}%
  \dsp@tabpause%
```

Now for some final setting up. The column separation is set from the user's parameter, the `\everycr` tokens are cleared, and I set up the newline command appropriately.

```
  \col@sep.5\eqainskip%
  \everycr{}%
  \let\\\@eqncr%
```

Now start a maths display and do the alignment. Set up the left hand tabskip glue to centre the alignment, and do the actual alignment. The preamble used is mainly that generated from the user's string, although the stuff at the end is how we set up the equation number – it repeats appropriately so we can always find it.

```
  \dsp@start%
  \tabskip\eqaopenskip%
  \halign to\displaywidth\expandafter\bgroup%
    \the\tab@preamble%
    &&\eqa@lastcol\hb@xt@\z@{\hss##}\tabskip\z@\cr%
}
```

Now for the end of the environment. This is really easy. Set the final equation number, close the `\halign`, tidy up the equation counter (it's been stepped once too many times) and close the display.

```
\def\endeqnarray{%
  \eqa@eqnum%
  \egroup%
  \dsp@end%
  \global\let\eqa@number\eqa@oldnumber%
  \global\@ignoretrue%
  \global\advance\c@equation\m@ne%
}
\expandafter\let\csname endeqnarray*\endcsname\endeqnarray
```

Now we can define the column types.

```
\colpush{eqnarray}
```

Note the positioning of ord atoms in the stuff below. This will space out relations and binops correctly when they occur at the edges of columns, and won't affect ord atoms at the edges, because ords pack closely.

First the easy onces. Just stick `\hfil` in the right places and everything will be all right.

```
\coldef r{\tabcoltype{\hfil$\eqastyle}{{}$}}
\coldef c{\tabcoltype{\hfil$\eqastyle{}}{{}$\hfil}}
\coldef l{\tabcoltype{$\eqastyle{}}{$\hfil}}
\coldef x{\tabcoltype{\if@fleqn\else\hfil\fi$\eqastyle}{$\hfil}}
```

Now for the textual ones. This is also fairly easy.

```
\collet T [tabular]T
```

Sort of split types of equations. I mustn't use `\rlap` here, or everything goes wrong – \\ doesn't get noticed by TEX in the same way as `\cr` does.

```
\coldef L{\tabcoltype{\hb@xt@2em\bgroup$\eqastyle}{$\hss\egroup}}
```

The '`:`' column type is fairly simple.

```
\coldef :{\tabspctype{\tabskip\eqacolskip}}
\coldef q{\tabspctype{\quad}}
```

The other column types just insert given text in an appropriate way.

```
\collet > [tabular]>
\collet < [tabular]<
\collet * [tabular]*
\collet @ [tabular]@
```

Finally, the magical '`\magic`' column type, which sets the equation number. We set up the `\tabskip` glue properly, tab on, and set the flag which marks the final column. The `\eqa@lastcol` command is there to raise an error if the user tabs over to this column. I'll temporarily redefine it to `\@eqalasttrue` when I enter this column legitimately. The extra magical bits here will make the final column repeat, so that we can find it if necessary. Well is this column type named.

That's it. We can return to normal now.

```
\colpop
```

3.4.2 Newline codes

Newline sequences (\\) get turned into calls of `\@eqncr`. The job is fairly simple, really.

```
\def\@eqncr{\tab@cr\eqacr@i\interdisplaylinepenalty\@M}%
\def\eqacr@i#1#2{%
  \eqa@eqnum%
  \noalign{\penalty#2\vskip\jot\vskip#1}%
}
```

3.4.3 Setting equation numbers

`\eqa@eqpos` Before we start, we need to generalise the flush-left number handling bits. The macro `\eqa@eqpos` will put its argument in the right place.

```
\if@leqno
  \def\eqa@eqpos#1{%
    \hb@xt@.01\p@{}\rlap{\normalfont\normalcolor\hskip-\displaywidth#1}%
  }
\else
  \def\eqa@eqpos#1{\normalfont\normalcolor#1}
\fi
```

`\eqa@eqnum` Here we typeset an equation number in roughly the right place. First I'll redefine `\eqa@lastcol` so that it tells me I'm in the right place, and start a loop to find that place.

```
\def\eqa@eqnum{%
  \global\let\eqa@lastcol\@eqalasttrue%
  \eqa@eqnum@i%
}
```

Now for the loop. The `\relax` here is absolutely vital – it starts the table column, inserting useful tokens like '`\eqa@lastcol`' which tell me where I am in the alignment. Then, if I've reached the end, I can typeset the equation number; otherwise I go off into another macro and step on to the next column.

```
\def\eqa@eqnum@i{%
  \relax%
  \if@eqalast%
    \expandafter\eqa@eqnum@ii%
  \else%
    \expandafter\eqa@eqnum@iii%
  \fi%
}
\def\eqa@eqnum@ii{%
  \eqa@eqpos\eqa@number%
  \global\let\eqa@number\eqa@defnumber%
  \global\let\eqa@lastcol\eqa@@lastcol%
  \cr%
}
\def\eqa@eqnum@iii{&\eqa@eqnum@i}
```

`\eqa@lastcol` This is used as a marker for the final column in an eqnarray environment. By default it informs the user that they've been very silly and swallows the contents of the column. I'll redefine it to something more useful at appropriate times, and then turn it back again.

```
\def\eqa@@lastcol{\mth@err@number\setbox\z@}
\let\eqa@lastcol\eqa@@lastcol
```

3.4.4 Numbering control

`\eqnumber` The `\eqnumber` command sets the equation number on the current equation. This is really easy, actually.

```
\newcommand\eqnumber[1][\eqa@eqcount]{\gdef\eqa@number{#1}}
```

`\eqa@eqcount` This is how a standard equation number is set, stepping the counter and all. It's really easy and obvious.

```
\def\eqa@eqcount{(\theequation)\global\advance\c@equation\@ne}
```

`\nonumber` The LaTeX `\nonumber` command could be defined by saying

```
\renewcommand{\nonumber}{\eqnumber[]}
```

but I'll be slightly more efficient and redefine `\eqa@number` directly.

```
\def\nonumber{\global\let\eqa@number\@empty}
```

3.4.5 The eqnalign environment

As a sort of companion to eqnarray, here's an environment which does similar things inside a box, rather than taking up the whole display width. It uses the same column types that we've already created, so there should be no problems.

`eqnalign` First, sort out some simple things like optional arguments.

```
\def\eqnalign{\@ifnextchar[\eqnalign@i{\eqnalign@i[rcl]}}
\def\eqnalign@i[#1]{%
  \@ifnextchar[{\eqnalign@ii{#1}}{\eqnalign@ii{#1}[c]}%
}
```

Now we actually do the environment. This is fairly easy, actually.

```
\def\eqnalign@ii#1[#2]{%
  \let\\\eqn@cr%
  \colset{eqnarray}%
  \tab@initread%
  \def\tab@tabtext{&\tabskip\z@skip}%
  \tabskip\z@skip%
  \col@sep.5\eqainskip%
  \tab@readpreamble{#1}%
  \everycr{}%
  \if#2t\vtop\else%
    \if#2b\vbox\else%
      \vcenter%
    \fi%
  \fi%
  \bgroup%
  \halign\expandafter\bgroup\the\tab@preamble\cr%
}
```

Finishing the environment is even simpler.

```
\def\endeqnalign{%
  \crcr%
  \egroup%
  \egroup%
}
```

`\eqn@cr` Newlines are really easy here.

```
\def\eqn@cr{\tab@cr\eqn@cr@i{}{}}
\def\eqn@cr@i#1{\cr\noalign{\vskip\jot\vskip#1}\@gobble}
```

3.5 Simple multiline equations

As a sort of example and abbreviation, here's a multiline display environment which just centres everything.

`eqlines` We just get `\eqnarray` to do everything for us. This is really easy.

```
\def\eqlines{\eqnarray[x]}
\let\endeqlines\endeqnarray
```

`eqlines*` There's a $*$ version which omits numbers. This is easy too. Lots of hacking with expansion here to try and reduce the number of tokens being used. Is it worth it?

```
\expandafter\edef\csname eqlines*\endcsname{%
  \expandafter\noexpand\csname eqnarray*\endcsname[x]%
}
\expandafter\let\csname endeqlines*\expandafter\endcsname
                \csname endeqnarray*\endcsname
```

3.6 Split equations

Based on an idea from *The TEXbook*, we provide some simple environments for doing split equations. These's plenty of scope for improvement here, though.

`spliteqn` `spliteqn*`

The only difference between these two is that the $*$-version doesn't put in an equation number by default (although this behaviour can be changed by `\eqnumber`).

The fun here mainly concerns putting in the equation number at the right place – for `leqno` users, we need to put the number on the first line; otherwise we put it on the last line.

The way we handle this is to have two macros, \\ (which clearly does all the user line breaks) and `\seq@lastcr` which is used at the end of the environment to wrap everything up. The `\seq@eqnocr` macro puts an equation number on the current line and then does a normal \\. It also resets \\ and `\seq@lastcr` so that they don't try to put another equation number in. This must be done globally, although anyone who tries to nest maths displays will get what they deserve.

For the non-$*$ environment, then, we need to step the equation counter, and set \\ to `\seq@cr` or `\seq@eqnocr` as appropriate for the setting of the `leqno` flag – `\seq@lastcr` always gets set to put an equation number in (because it will be reset if the number actually gets done earlier – this catches stupid users trying to put a single row into a split environment).

```
\def\spliteqn{%
  \let\eqa@oldnumber\eqa@number%
  \global\let\eqa@number\eqa@eqcount%
  \spliteqn@i%
}
```

For the $*$ variant, we don't need to bother with equation numbering, so this is really easy.

```
\@namedef{spliteqn*}{%
  \let\eqa@oldnumber\eqa@number%
  \gdef\eqa@number{}%
  \spliteqn@i%
}
```

Ending the environments is easy. Most of the stuff here will be described later.

```
\def\endspliteqn{%
  \hfilneg\seq@lastcr%
  \egroup%
  \dsp@end%
  \global\let\eqa@number\eqa@oldnumber%
  \global\advance\c@equation\m@ne%
}
\expandafter\let\csname endspliteqn*\endcsname\endspliteqn
```

`\spliteqn@i`

Here we handle the full display splits. Start a maths display, and make each row of the alignment take up the full display width.

The macro `\seq@dosplit` does most of the real work for us – setting up the alignment and so forth. The template column is interesting. There are two items glue on both sides of the actual text:

- Some glue which can shrink. This keeps the display from the edges of the page unless we get a really wide item.

- An `\hfil` to do the alignment. By default, this centres the equations. On the first line, however, we put a leading `\hfilneg` which cancels the first `\hfil`, making the first row left aligned. Similarly, at the end, we put an `\hfilneg` after the last equation to right align the last line.

We pass this information on as an argument. It's easy really.

```
\def\spliteqn@i{%
```

First, set up equation numbering properly. See my rant about `\refstepcounter` above.

```
  \stepcounter{equation}%
  \def\@currentlabel{\p@equation\theequation}%
```

Right; now to sort out the numbering and newline handling. If the number's meant to be on the first line (for leqno users), then it gets typeset on the first like; otherwise we just do a normal newline on all lines except the first. Once `\seq@eqnocr` has done its stuff, it redefines all the newline handling not to insert another number.

```
  \if@leqno%
    \global\let\seq@docr\seq@eqnocr%
  \else%
    \global\let\seq@docr\seq@cr%
  \fi%
  \global\let\seq@lastcr\seq@eqnocr%
```

For my next trick, I'll do some display handling – start a (possibly nested) maths display, set up the `\tabpause` macro appropriately, and set the newline command to do the right thing.

```
  \dsp@start%
  \dsp@tabpause%
  \def\\{\seq@docr}%
```

Finally, call another macro to do the remaining bits of setting up.

```
  \seq@dosplit%
    {\hb@xt@\displaywidth{%
      \hskip\splitleft\hfil$\displaystyle##$%
      \hfil\hskip\splitright}}%
    {\hfilneg}%
}
```

`subsplit` For doing splits in the middle of equations, we provide a similar environment. Here, we make `\\` just start a new line. We also use a `\vcenter` rather than a full maths display. The glue items are also a bit different: we use plain double-quads on each side of the item, and we need to remove them by hand at the extremeties of the environment.

```
\def\subsplit{%
  \let\\\seq@cr%
  \vcenter\bgroup%
  \seq@dosplit{\hfil\qquad$##$\qquad\hfil}{\hfilneg\hskip-2em}%
}
```

Ending the environment is fairly easy. We remove the final glue item, and close the alignment and the vbox.

```
\def\endsubsplit{%
  \hfilneg\hskip-2em\cr%
  \egroup\egroup%
}
```

`\seq@dosplit` Here we do most of the real work. Actually, since the preamble is passed in as an argument, most of the work is already done. The only thing to really note is the template for subsequent columns. To stop users putting in extra columns (which is where we put the equation number) we raise an error and discard the input in a scratch box register. This template is repeated infinitely so as to allow us to put the equation number in nicely. However, the final negative glue item won't work properly, so the equation will look awful.

```
\def\seq@dosplit#1#2{%
  \halign\bgroup%
    #1&&\mth@err@number\setbox\z@\hbox{##}\cr%
  #2\relax%
}
```

`\seq@eqnocr` Here's how we set equation numbers. Since the column provided raises errors as soon as a token finds its way into it, we start with a `&\omit`. Then we just put the equation number in a zero-width box. Finally, we reset the newline commands to avoid putting in more than one equation number, and do normal newline things.

```
\def\seq@eqnocr{%
  &\omit%
  \hb@xt@\z@{\hss\eqa@eqpos\eqa@number}%
  \global\let\seq@docr\seq@cr%
  \global\let\seq@lastcr\seq@cr%
  \seq@cr%
}
```

`\seq@cr` Newlines are very easy. We add a `\jot` of extra space, since this is a nice thing to do.

```
\def\seq@cr{\tab@cr\seq@cr@i\interdisplaylinepenalty\@M}
\def\seq@cr@i#1#2{\cr\noalign{\penalty#2\vskip\jot\vskip#1}}
```

3.7 Matrix handling

There's been a complete and total overhaul of the spacing calculations for matrices here. The vertical spacing now bears an uncanny similarity to the rules TEX uses to space out `\atop`-like fractions, the difference being that you can have more than one column in a matrix. This has the interesting side-effect that we get an amsmath-style sub/superscript environment almost free of charge with the matrix handling (it just ends up being a script-size single-column matrix).

What is rather gratifying is that our matrix environment looks rather nicer than amsmath's (which is based directly on array, giving it nasty restrictions on the numbers of columns and so on); in particular, the version here gives the 'correct' result for Knuth's exercise 18.42 (which states categorically that a `\smallskip` should be placed between the rows of the big matrix).

The reason the interrow space doesn't come out in the AMS version is that array inserts extra vertical space by extending the depth of the final row using a strut: the big matrix already extends deeper than this, so the strut doesn't make any difference. If the space was added by `\hlx{s[\smallskipamount]}` instead of the \\ command, things would be different.

Exercise 18.42 from *The TEXbook*

$$\begin{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} & \begin{pmatrix} e & f \\ g & h \end{pmatrix} \\ 0 & \begin{pmatrix} i & j \\ k & l \end{pmatrix} \end{pmatrix}$$

$$\begin{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} & \begin{pmatrix} e & f \\ g & h \end{pmatrix} \\ 0 & \begin{pmatrix} i & j \\ k & l \end{pmatrix} \end{pmatrix}$$

```
\newcommand{\domatrix}[1]{
  \def\mat##1
    {\begin{#1}##1\end{#1}}
  \[ \begin{#1}
      \mat{a & b \\ c & d} &
      \mat{e & f \\ g & h}
      \\[\smallskipamount]
      0 &
      \mat{i & j \\ k & l}
      \end{#1}
  \]
}
\domatrix{pmatrix}
\domatrix{ams-pmatrix}
```

genmatrix The first job is to store my maths style and font away, because I'll be needing it lots later.

```
\def\genmatrix#1#2#3#4#5{%
  \let\mat@style#1%
  \ifx#2\scriptstyle%
    \let\mat@font\scriptfont%
  \else\ifx#2\scriptscriptstyle%
    \let\mat@font\scriptscriptfont%
  \else%
    \let\mat@font\textfont%
  \fi\fi%
```

Now to cope with inserted text. This is easy.

```
  \ifx\mat@style\scriptstyle%
    \let\mat@textsize\scriptsize%
  \else\ifx\mat@style\scriptscriptstyle%
    \let\mat@textsize\scriptscriptsize%
  \else%
    \let\mat@textsize\relax%
  \fi\fi%
```

Now for some fun. I'll remember how to start and end the matrix in a couple of macros `\mat@left` and `\mat@right`. I haven't yet worked out exactly what needs to be in `\mat@right` yet, though, so I'll build that up in a scratch token list while I'm making my mind up.

Initially, I want to open a group (to trap the style changes), set the maths style (to get the right spacing), insert the left delimiter, insert some spacing around

the matrix, and start a centred box. The ending just closes all the groups and delimiters I opened.

```
\def\mat@left{\bgroup\mat@style\left#4#3\vcenter\bgroup}%
\toks@{\egroup#3\right#5\egroup}%
```

Now comes a slightly trickier bit. If the maths style is script or scriptscript, then I need to raise the box by a little bit to make it look really good. The right amount is somewhere around $^3/_4$ pt, I think, so that's what I'll use.

```
\@tempswatrue%
\ifx\mat@style\displaystyle\else\ifx\mat@style\textstyle\else%
  \@tempswafalse%
  \setbox\z@\hbox\bgroup$%
  \toks@\expandafter{\the\toks@$\m@th\egroup\raise.75\p@\box\z@}%
\fi\fi%
```

If I'm not in maths mode right now, then I should enter maths mode, and remember to leave it later.

```
\if@tempswa\ifmmode\else%
  $\m@th%
  \toks@\expandafter{\the\toks@$}%
\fi\fi%
```

Now I've sorted out how to end the environment properly, so I can set up the macro, using `\edef`.

```
\edef\mat@right{\the\toks@}%
```

Now see if there's an optional argument. If not, create lots of centred columns.

```
\@ifnextchar[\genmatrix@i{\genmatrix@i[[c]]}%
}
```

Now to sort out everything else.

```
\def\genmatrix@i[#1]{%
```

Some initial setting up: choose the correct column set, and set up some variables for reading the preamble.

```
\colset{matrix}%
\tab@initread%
```

Now comes some of the tricky stuff. The space between columns should be 12 mu (by trial and error). We put the space in a box so we can measure it in the correct mathstyle.

```
\setbox\z@\hbox{$\mat@style\mskip12mu$}%
\edef\tab@tabtext{&\kern\the\wd\z@}%
\tab@readpreamble{#1}%
```

Now we need to decide how to space out the rows. The code here is based on the information in appendix G of *The TEXbook*: I think it'd be nice if my matrices were spaced out in the same way as normal fractions (particularly `\choose`y things). The standard `\baselineskip` and `\lineskip` parameters come in really handy here.

The parameters vary according to the size of the text, so I need to see if we have scriptsize or less, or not. The tricky `\if` sorts this out.

```
\if1\ifx\mat@style\scriptstyle1\else%
```

```
      \ifx\mat@style\scriptscriptstyle1\else0\fi\fi%
    \baselineskip\fontdimen10\mat@font\tw@%
    \advance\baselineskip\fontdimen12\mat@font\tw@%
    \lineskip\thr@@\fontdimen8\mat@font\thr@@%
  \else%
    \baselineskip\fontdimen8\mat@font\tw@%
    \advance\baselineskip\fontdimen11\mat@font\tw@%
    \lineskip7\fontdimen8\mat@font\thr@@%
  \fi%
  \lineskiplimit\lineskip%
```

Now actually set up for the alignment. Assign \\ to the correct value. Set up the `\tabskip`. Do the appropriate `\mat@left` thing set up above. And then start the alignment.

```
  \let\\\mat@cr%
  \tabskip\z@skip%
  \col@sep\z@%
  \mat@left%
  \halign\expandafter\bgroup\the\tab@preamble\tabskip\z@skip\cr%
```

Now for a little hack to make the spacing consistent between matrices of the same height. This comes directly from Plain TeX. This appears to make the spacing *exactly* the same as the TeX primites, oddly enough.

```
  \ifx\mat@font\textfont%
    \omit$\mat@style\mathstrut$\cr\noalign{\kern-\baselineskip}%
  \fi%
}
```

Finishing the environment is really easy. We do the spacing hack again at the bottom, close the alignment and then tidy whatever we started in `\mat@left`.

```
\def\endgenmatrix{%
  \crcr%
  \ifx\mat@font\textfont%
    \omit$\mat@style\mathstrut$\cr\noalign{\kern-\baselineskip}%
  \fi%
  \egroup%
  \mat@right%
}
```

`\mat@cr` Newlines are really easy. The $*$-form means nothing here, so we ignore it.

```
\def\mat@cr{\tab@cr\mat@cr@i{}{}}
\def\mat@cr@i#1{\cr\noalign{\vskip#1}\@gobble}
```

`\newmatrix` This is how we define new matrix environments. It's simple fun with `\csname` and `\expandafter`.

```
\def\newmatrix#1#2{%
  \@namedef{#1}{\genmatrix#2}%
  \expandafter\let\csname end#1\endcsname\endgenmatrix%
}
```

`matrix` `pmatrix` `dmatrix` `smatrix` `spmatrix` `sdmatrix` `smatrix*` `spmatrix*` `sdmatrix*` Now we define all the other environments we promised. This is easy.

```
\newmatrix{matrix}{{\textstyle}{\textstyle}{\,}{.}{.}}
\newmatrix{pmatrix}{{\textstyle}{\textstyle}{\,}{(}{)}}
```

```
\newmatrix{dmatrix}{{\textstyle}{\textstyle}{\,}}
\newmatrix{smatrix}{{\scriptstyle}{\scriptstyle}{}{.}{.}}
\newmatrix{spmatrix}{{\scriptstyle}{\scriptstyle}{}{(}{)}}
\newmatrix{sdmatrix}{{\scriptstyle}{\scriptstyle}{}}
\newmatrix{smatrix*}{{\scriptstyle}{\textstyle}{}{.}{.}}
\newmatrix{spmatrix*}{{\scriptstyle}{\textstyle}{}{(}{)}}
\newmatrix{sdmatrix*}{{\scriptstyle}{\textstyle}{}}
```

`script` Now for superscripts and subscripts. This is fairly easy, because I took so much care over the matrix handling.

```
\def\script{%
  \let\mat@style\scriptstyle%
  \def\mat@left{\vcenter\bgroup}%
  \def\mat@right{\egroup}%
  \let\mat@font\scriptfont%
  \let\mat@textsize\scriptsize%
  \@ifnextchar[\genmatrix@i{\genmatrix@i[c]}%
}
\let\endscript\endgenmatrix
```

Now define the column types.

```
\colpush{matrix}
\coldef l{\tabcoltype{\kern\z@$\mat@style}{\m@th$\hfil}}
\coldef c{\tabcoltype{\hfil$\mat@style}{\m@th$\hfil}}
\coldef r{\tabcoltype{\hfil$\mat@style}{\m@th$}}
\coldef T#1{\tab@aligncol{#1}{\begingroup\mat@textsize}{\endgroup}}
```

The repeating type is more awkward. Things will go wrong if this is given before the first column, so we must do a whole repeat by hand. We can tell if we haven't contributed a column yet, since `\tab@column` will be zero. Otherwise, we fiddle the parser state to start a new column, and insert the `&` character to make TEX repeat the preamble.

```
\coldef {[]{%
  \@firstoftwo{%
    \ifnum\tab@columns=\z@%
      \def\@tempa##1\q@delim{%
        \tab@mkpreamble##1[##1\q@delim%
      }%
      \expandafter\@tempa%
    \else%
      \tab@setstate\tab@prestate%
      \tab@append\tab@preamble{&}%
      \expandafter\tab@mkpreamble%
    \fi%
  }%
}
```

We're done defining columns now.

```
\colpop
```

3.8 Dots...

Nothing whatsoever to do with alignments, although vertical and diagonal dots in small matrices look really silly. The following hacky definitions work rather better.

`\mdw@dots` First of all, here's some definitions common to both of the dots macros. The macro takes as an argument the actual code to draw the dots, passing it the scaled size of a point in the scratch register `\dimen@`; the register `\box 0` is set to contain a dot of the appropriate size.

```
\def\mdw@dots#1{\ensuremath{\mathpalette\mdw@dots@i{#1}}}
\def\mdw@dots@i#1#2{%
  \setbox\z@\hbox{$#1\mskip1.8mu$}%
  \dimen@\wd\z@%
  \setbox\z@\hbox{$#1.$}%
  #2%
}
```

`\vdots` I'll start with the easy one. This is a simple translation of the original implementation.

```
\def\vdots{%
  \mdw@dots{\vbox{%
    \baselineskip4\dimen@%
    \lineskiplimit\z@%
    \kern6\dimen@%
    \copy\z@\copy\z@\box\z@%
  }}%
}
```

`\ddots` And I'll end with the other easy one...

```
\def\ddots{%
  \mdw@dots{\mathinner{%
    \mkern1mu%
    \raise7\dimen@\vbox{\kern7\dimen@\copy\z@}%
    \mkern2mu%
    \raise4\dimen@\copy\z@%
    \mkern2mu%
    \raise\dimen@\box\z@%
    \mkern1mu%
  }}%
}
```

3.9 Lucky dip

Time to round off with some trivial environments, just to show how easy this stuff is.

`cases` These are totally and utterly trivial.

`smcases`

```
\def\cases{\left\{\,\array{@{}lTl@{}}}
\def\endcases{\endarray\,\right.}
\def\smcases{\left\{\smarray{@{}lTl@{}}}
\def\endsmcases{\endsmarray\,\right.}
```

3.10 Error messages

Some token saving:

```
\def\mth@error{\PackageError{mathenv}}
```

Now for the error messages.

```
\def\mth@err@number{%
  \mth@error{Too many `&' characters found}{%
    You've put too many `&' characters in an alignment^^J%
    environment (like `eqnarray' or `spliteqn') and wandered^^J%
    into trouble.  I've gobbled the contents of that column^^J%
    and hopefully I can recover fairly easily.%
  }%
}
\def\mth@err@mdsp{%
  \mth@error{Can't do displays in nondisplay maths mode}{%
    You're trying to start a display environment, but you're^^J%
    in nondisplay maths mode.  The display will appear but^^J%
    don't blame me when it looks horrible.%
  }%
}
\def\mth@err@hdsp{%
  \mth@error{Can't do displays in LR mode}{%
    You're trying to start a display environment, but you're^^J%
    in LR (restricted horizontal) mode.  Everything will go^^J%
    totally wrong, so your best bet is to type `X', fix the^^J%
    mistake and start again.%
  }%
}
```

That's all there is. Byebye.

```
⟨/mathenv⟩
```

Mark Wooding, 28 April 1998

Appendix

A The GNU General Public Licence

The following is the text of the GNU General Public Licence, under the terms of which this software is distrubuted.

A.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

A.2 Terms and conditions for copying, distribution and modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do

not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. **Because the Program is licensed free of charge, there is no warranty for the Program, to the extent permitted by applicable law. except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Program is with you. Should the Program prove defective, you assume the cost of all necessary servicing, repair or correction.**

12. **In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the Program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.**

END OF TERMS AND CONDITIONS

A.3 Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of

warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

Numbers written in italic refer to the page where the corresponding entry is described, the ones underlined to the code line of the definition, the rest to the code lines where the entry is used.

T

V