Warning!

This style option is in the early stages of development. If you want to use an extended \texttt{array} or \texttt{tabular} in a document, consider using one of the options in the ARRAY package, available from most \TeX-server.

The commands defined in this style are quite likely to have both their user-interface, and their internal definitions changed in later versions.

1 Introduction

This style option implements an environment, \texttt{blockarray}, that may be used in the same way as the \texttt{array} or \texttt{tabular} environments of standard \LaTeX, or their extended versions defined in \texttt{array.sty}. If used in math-mode, \texttt{blockarray} acts like \texttt{array}, otherwise it acts like \texttt{tabular}.

The main feature of this style is that it uses a new method of defining column types. In the simplest form, this has been given a syntax matching the \texttt{\newcolumntype} command of \texttt{array.sty}.

\begin{verbatim}
\newcolumntype{C}{>{\large}c}
\end{verbatim}

defines a column of large centred text.

In \texttt{array.sty} column specifiers defined via \texttt{\newcolumntype} are re-written in a preliminary stage to the primitive types, which are then treated by a completely different mechanism (basically a nested \texttt{\if} testing each token against one of the predefined column types, \texttt{c, l, >, ...}

In \texttt{blockarray.sty}, all column specifiers have equal standing, most of the specifiers of Lamport’s original are defined using \texttt{\newcolumntype}, e.g.

\begin{verbatim}
\newcolumntype{c}{>{\hfil}<\hfil}
\end{verbatim}

There are one or two other features built into \texttt{blockarray}, these will be introduced in no particular order.

1.1 Explicit Column Separators in the Preamble

As described in the \LaTeX~book, originally specifiers like | and \texttt{\&}-expressions were always considered to be part of the \textit{preceding} column (except for expressions before
the first column). This can be inconvenient if that column type is going to be over
ridden by a \multicolumn specification, consider:
\begin{tabular}{c|c|c}
  11 & 22 & 33 \\
  1 & \multicolumn{1}{l|}{2} & 3 \\
  11 & 22 & 33
\end{tabular}

The rule needs to be specified again in the \multicolumn argument as \{1\},
\blockarray lets you move the rule into the third column, by specifying & in the
preamble like so:
\begin{blockarray}{c|c&|c}
  11 & 22 & 33 \\
  1 & \BAmulticolumn{1}{l}{2} & 3 \\
  11 & 22 & 33
\end{blockarray}

I first came across the idea of having such a feature in an array preamble
when Rainer Schöpf gave a brief introduction to various enhanced array styles.
An implementation by Denys Duchier had a feature like this, however I have not
seen that style so I do not know the details.

1.2 Blocks

Sometimes you want whole blocks of the table to have a different format, this is
often the case with headings for instance. This can be accomplished using lots
of \multicolumn commands, but this style lets you specify the format for such a
block in the usual syntax for column specifiers:
\begin{blockarray}{*{3}{c}}
  11 & 22 & 33 \\
  1 & 2 & 3 \\
\begin{block}{*{3}{>{\bf}l}}
  11 & 22 & 33 \\
  1 & 2 & 3 \\
\end{block}
\end{blockarray}

1.3 Delimiters

People often want to put delimiters around sub-arrays of a larger array, delimiters
can now be specified in the preamble argument:
\begin{blockarray}{(cc)c}
11 & 22 & 33 \\
1 & 2 & 3 \\
\end{blockarray}

\begin{block}{(l)l}
11 & 22 & 33 \\
1 & 2 & 3 \\
\end{block}

\begin{blockarray}{(l)c}
11 & 22 & 33 \\
1 & 2 & 3 \\
\end{blockarray}

Note how in the previous example the nested block was not spanned by the \[\]. each section of the 'outer' block was separately bracketed. If instead of the block environment we use block*, then the outer brackets will span the inner block, however it is not possible to specify any delimiters in the argument of block*.

\begin{blockarray}{(cc)c}
11 & 22 & 33 \\
1 & 2 & 3 \\
\end{blockarray}

The delimiters, ( ) [ ] \{ \} have been pre-defined as column specifiers, however any delimiter, including these ones can be specified using the specifiers \Left and \Right \Left\{text\}\{delimiter\} specifies a delimiter together with a 'label' which will be vertically centred with respect to the block. Note that the delimiter and the label take up no horizontal space, and so extra space must be left with a !- or @-expression or the text will over-print adjacent columns.

1.4 Automatic Numbering

A column specifier \BAenum specifies that the row number is to be printed (in a !-expression) at that point in each row, this number may be accessed with \label in the usual way. The number is a standard L\TeX counter, \BAenumi, and so the appearence may be changed by altering the default definition of \the\BAenumi.

1.5 Footnotes

The \footnote command may be used inside blockarray. Two styles are supported. If the test \BAtablenotes is true (by default, or after \BAtablenotestrue) then footnotes will appear at the end of the table, with lines set to the width of the table. If \BAtablenotes is false, footnotes within the table will be treated as standard footnotes, with the text (usually) appearing at the foot of the page.

If table notes are being set, the footnote counter is reset at the start of the table. Also an extended version of \footnotetext can be used. As described in the book,
\footnotetext[2]{xxx} will produce a text marked with the footnote symbol for ‘2’. However for tablenotes, the optional argument may also be any non-numeric text, in which case it is set directly at the start of the footnote text. So you can go \footnotetext[\sc source:]{xxx} or \footnotetext[\sc note:]{xxx} anywhere in the table body, before the first numbered footnote.

If \texttt{\textbackslash BAtablenotes} is false the footnote text will not appear at the foot of the page if the whole \texttt{blockarray} environment is in an environment which treats footnotes in a special way (eg another \texttt{blockarray}). So if you have a complicated table which requires tablenotes, but for \TeX{}nical reasons you wish to enter it in the \texttt{.tex} file as nested \texttt{blockarray} environments, you may set \texttt{\textbackslash BAtablenotestrue} for the outer environment, and then locally set it to false before each of the nested environments. This will ensure that footnotes from all parts of the table will be collected together at the end.

This table is set with \texttt{\textbackslash BAtablenotestrue}.

\begin{tabular}{|l|l|l|l|}
\hline
& ONE & & TWO* \\
& l-one & l-two & r-one & r-two \\
& l-three* & l-four & r-three* & r-four \\
\hline
\end{tabular}

\texttt{source:} Chicago Manual of Style.  
\texttt{note:} The above attribution is incorrect.  
\texttt{* Footnote to l-three.} 
\texttt{THREE†} 
\texttt{note:} The above attribution is incorrect.  
\texttt{* Footnote to r-three} 
\texttt{FOUR} 

In this example, the outer table is set with \texttt{\textbackslash BAtablenotestrue}, but each of the inner tables is set with a local setting \texttt{\textbackslash BAtablenotesfalse}.

Also the footnotes have been set in a single paragraph. Tablenotes will be set ‘run in’ a paragraph, after a \texttt{\textbackslash BAparrfootnotes} declaration.

\begin{tabular}{|l|l|l|l|}
\hline
& ONE & & TWO* \\
& l-one & l-two & r-one & r-two \\
& l-three† & l-four & r-three‡ & r-four \\
\hline
\end{tabular}

\texttt{source:} Chicago Manual of Style.  
\texttt{note:} The above attribution is incorrect.  
\texttt{* Note on TWO. This is a reasonably long footnote, to show the line breaking.} 
\texttt{†Footnote to l-three.} 
\texttt{‡Footnote to r-three} 
\texttt{§Note on THREE.
1.6 Non Aligned Material

The primitive \noalign command may be used with standard \LaTeX arrays, but paragraphs inside \noalign are broken into lines that are the width of the page, (or at least the current value of \hsize) not to the final width of the table. Within a blockarray \blockarray specifies material to be packaged into a parbox the same width as the table. This makes a ‘hole’ in the current block. \blockarray* is similar, but any delimiters in the current block span across the non-aligned paragraphs.

\begin{blockarray}{\blockarray!{.quad}cc\Right{}}
\blockarray*{... The paragraphs ...}
\begin{blockarray}{\blockarray!{.quad}(rr\Right{}}\{tt block 2} ......}
\begin{blockarray}{\blockarray!{.quad}(ll)}
\begin{blockarray}{\blockarray!{.quad}>{\bf}l\{c\Right{}}block 3}
\begin{block}{\blockarray!{.quad}}block 4
\begin{block}{\blockarray!{.quad}\centering block 1
1. ccc cc
2. c cccccc
The paragraphs in a \blockarray* are set to the final width of the table.
3. ( rrr rr \\
4. rrr r \\
5. lll l \\
6. l lll \\
7. r r \\
8. ccc \\
9. LLL Spanning all the rows in a block.
10. LLL \\
11. L \\
12. L \\
13. L \\
14. c c}

Unlike \blockarray*, \blockarray breaks any delimiters in the current block.

\blockarray 1
\blockarray 2
\blockarray 3
\blockarray 4
\blockarray 1
5
1.7 Spanning Rows and Columns

The previous table had an example of the use of a \texttt{\LaTeX} command. If an entry contains
\texttt{\LaTeX\multirow{⟨dimen⟩}{⟨par-mode material⟩}}
then the ⟨par-mode material⟩ will appear in a box at least ⟨dimen⟩ wide, spanning all the rows in the current block. If the other entries in that column of the current block are not empty, they will be over printed by the spanning text.

There is a column specification corresponding to \texttt{\LaTeX\multirow{⟨dimen⟩}}. if \texttt{\LaTeX\multirow{⟨dimen⟩}} appears in the preamble, then each entry in that column will be packaged as a paragraph, in a box at least ⟨dimen⟩ wide, spanning all the rows in the current block. If this is the last column in the block, you cannot use the optional argument to \textbackslash\, and no entry in the column must be empty, it must at least have {} in it. (If you need to ask why, you don’t want to know!)

Similarly there is a column specification corresponding to \texttt{\LaTeX\multicolumn{⟨number⟩}{⟨column specification⟩}}. if \texttt{\LaTeX\multicolumn{⟨number⟩}{⟨column specification⟩}} appears in the preamble to a block, then the rows in the block should have less entries than the outer block, the columns will line up as expected.

\begin{blockarray}{r|lccr|c}
\begin{block}{(r|\LaTeX\multicolumn{4}{>{\bf}l}|c)}  
111&The second entry in each &333\end{blockarray}
\end{blockarray}
\begin{blockarray}{r|lccr|c}
\begin{block}{[r|lccr}{\LaTeX\multirow{1in}}}
111&222&333&444&555&Each entry\end{blockarray}
\end{blockarray}

\begin{blockarray}{r|lccr|c}
\begin{block}{[r|lccr}{\LaTeX\multirow{1in}}}
1&2&3&4&5&in this column is packaged as a paragraph.\end{blockarray}
\end{blockarray}
\begin{blockarray}{r|lccr|c}
\begin{block}{[r|lccr}{\LaTeX\multirow{1in}}}
1&2&3&4&5&\relax\end{blockarray}
\end{blockarray}
\begin{blockarray}{||c||c|cc||cc||}
\BAhline{|t:=:t:=&|==#==:t|}
0 & 1 & 2 & 3 & 4 & 5 \\
\BAline
\BAhline 0 & 1 & 2 & 3 & 4 & 5 \\
\BAline
\BAhline 0 & 1 & 2 & 3 & 4 & 5 \\
\BAhline 0 & 1 & 2 & 3 & 4 & 5 \\
\BAhline
\BAhline{=}::==:}
0 & 1 & 2 & 3 & 4 & 5 \\
\BAline
\BAhline{b:=:b:===b:}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{blockarray}

Both \texttt{\hline} and \texttt{\hhline} increase the (minimum) height of the following row by \texttt{\extrarowheight}, which defaults to 0pt. \texttt{array.sty} introduced a parameter, known in this style as \texttt{\extrarowheight}, which is a length added to the default height of all the rows. One of the stated reasons for introducing this was to stop horizontal lines touching large entries like accented capitals, however increasing all the row heights has an effect rather similar to setting \texttt{\arraystretch}. This style allows the extra height just to be added after the horizontal rule.

### 1.8 Horizontal Lines

For technical reasons (explained in the code section) the standard \texttt{\hline} does not work with \texttt{blockarray}. \texttt{\BAhline} may be used in just the same way, although currently it is implemented using...

\texttt{\BAhline}. The \texttt{\hline} from \texttt{hhline.sty}, would work, but this is a new implementation, more in the spirit of this style.

\begin{verbatim}
\begin{blockarray}{||c||c|cc||cc||}
\BAhline{|t:=:t:=&|==#==:t|}
0 & 1 & 2 & 3 & 4 & 5 \\
\BAline
\BAhline 0 & 1 & 2 & 3 & 4 & 5 \\
\BAline
\BAhline 0 & 1 & 2 & 3 & 4 & 5 \\
\BAline
\BAhline 0 & 1 & 2 & 3 & 4 & 5 \\
\BAhline{=}::==:}
0 & 1 & 2 & 3 & 4 & 5 \\
\BAline
\BAhline{b:=:b:===b:}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{blockarray}
\end{verbatim}
1.9 Further Thoughts

• The main point of any environment based on \halign is to make entries line up. Using this style as currently implemented, it is easy to spoil this alignment by putting different \( \diamond \) expressions or rules in the same column in different blocks. In practice, if you want different \( \diamond \) expressions, you need to do boxing tricks to make sure that they all have the same width. This could be done automatically by the \halign, if the \( \diamond \)-expressions and rules were put in a separate column. (This fact could be hidden from the user, by a method similar to the multicolumn column specification).

• The \( \text{[tcb]} \) optional argument does not really work at present, I have not done a proper implementation, as I do not know what to do about horizontal rules.

Standard \LaTeX\ lines \( \text{[t]} \) and \( \text{[b]} \) up like this:

\[
\begin{array}{|c|c|c|}
\hline
1 & xx & 3 \\
2 & & \\
3 & & \\
\hline
\end{array}
\]

However if there are horizontal lines, it looks like:

\[
\begin{array}{|c|c|c|}
\hline
1 & xx & \\
2 & & \\
3 & & \\
\hline
\end{array}
\]

I think I want it to look like:

\[
\begin{array}{|c|c|c|}
\hline
1 & xx & 3 \\
2 & & \\
3 & & \\
\hline
\end{array}
\]

This would be reasonably easy to achieve in a ‘full’ blockarray, as each row is taken off and inspected, however I would like an array that only uses the features of the original implementation to be processed by the ‘quick’ system. Any ideas?

• Many user-level commands and parameters defined in this style are named \( \text{BA} \ldots \) This is to avoid clashes with the standard environments, especially if these are nested inside blockarray. If array and tabular were re-defined in terms of blockarray, many commands could be renamed, for example, \( \text{BAextrarowheight, BAmulticolumn, BAhline} \).

• This style uses a lot of macros, and every use of the blockarray uses a lot more. Does it work at all on a PC?
2 The Macros

\ProvidesPackage{blkarray}[2015/02/27 v0.07 Block array (dpc)]

2.1 Some General Control Macros

The macros in this section do not have the \texttt{BA} prefix, but rather the \texttt{GC} prefix, other style files can repeat these definitions without using up \TeX’s memory.

\LaTeX\ provides \texttt{\@}, \texttt{\@ne}, \texttt{\tw@}, \texttt{\thr@@}, but I needed some more…

\begin{verbatim}
\chardef\GC@four=4
\chardef\GC@five=5
\chardef\GC@six=6
\end{verbatim}

2.1.1 Tests

Tests are like \texttt{\if\else\fi} except that instead of the \texttt{\if\ldots\else\ldots\fi} notation, they have:

\texttt{\test\ldots\{\true\ldots\}\{\false\ldots\}}

They are constructed such that they expand directly to either the \texttt{\true} or \texttt{\false}, without leaving a trailing \texttt{\fi}.

\begin{verbatim}
\def\GC@newtest#1{%
  \@namedef{#1true}{\expandafter\let\csname test#1\endcsname\GC@true}%
  \@namedef{#1false}{\expandafter\let\csname test#1\endcsname\GC@false}%
  \@nameuse{#1false}}%
\def\GC@def@testfromif#1#2\fi{%
  \def#1##1##{#2##1\expandafter\GC@true\else\expandafter\GC@false\fi}}%
\def\GC@true#1#2{#1}%
\def\GC@false#1#2{#2}
\end{verbatim}

This \texttt{\test\GC@num} is not very good as it does not delimit the \texttt{number}s correctly.

\begin{verbatim}
\GC@def@testfromif\test\GC@x\ifx\fi
\GC@def@testfromif\test\GC@num\ifnum\fi
\end{verbatim}

2.1.2 List Macros

If \texttt{X} is \texttt{abc} then \texttt{\GC@add@to@front\X{xyz}} is \texttt{xyzabc};

\begin{verbatim}
\long\def\GC@add@to@front#1#2{%
  \def\@tempa##1{\gdef#1{#2##1}}%
  \expandafter\@tempa\expandafter{#1}}%
\end{verbatim}

and \texttt{\GC@add@to@end\X{xyz}} is \texttt{abcxyz}.

\begin{verbatim}
\long\def\GC@add@to@end#1#2{\expandafter\gdef\expandafter#1\expandafter{#1#2}}%
\end{verbatim}

\begin{verbatim}

9
\end{verbatim}
2.2 Allocations

I have given ‘meaningful names’ to plain-T\TeX{}’s scratch registers, I am not sure this was a good idea, but it should be OK as long as I always access by name, and do not use, say, \texttt{\count4} as a scratch register. I do not like using numbered registers in the code, and can not afford to allocate registers just to get nice names, they are in too short supply already!

Only allocate another register if \texttt{blockarray} is going to lose control at a point where the register value needs to be saved. (eg inside a \texttt{\BAnoalign} anything can happen.

\texttt{\BAt tracing} can be set to any integer, the higher the number, the more gets printed.

\begin{verbatim}
22 \+\texttt{tracing}\+
23 \chardef\BAt tracing=0
24 \+\texttt{\textbackslash tracing}\+
25 \newcounter{BAenumi}\+\texttt{let} BA\texttt{@row}c\+\texttt{BAenumi}
26 \countdef\BA@row@shadow=6
27 \countdef\BA@ftn@shadow=0
28 \newcount\BA@col
29 \countdef\BA@col@shadow=2
30 \newcounter\BA@block@cnt
31 \countdef\BA@block@cnt@shadow=4
32 \countdef\BA@col@max=8
33 \newbox\BA@final@box
34 \chardef\BA@final@box@shadow=8
 0.07 use newbox not 0 to avoid amsmath.
35 \newbox\BA@first@box
36 \chardef\BA@tempbox@a=2
37 \chardef\BA@tempbox@b=4
38 \chardef\BA@block@box=6
39 \newdimen\BA@colsep
40 \BA@colsep=\tabcolsep
41 \newtoks\BA@ftn
42 \toksdef\BA@ftnx@shadow=0
\end{verbatim}

2.3 ‘Local’ Variables

Most of \texttt{blockarray} happens inside a \texttt{\halign} which means that the different parts have to make global assignments if they need to communicate. However many of these assignments are logically local to \texttt{blockarray}, or a sub-environment like \texttt{block}. This means that I have to manage the saving and restoring of local values ‘by hand’.

Three different mechanisms occured to me, I have used all of them in this style, mainly just to try them out!
• 'shadowing' If $X$ is to be assigned globally, but it is to be considered local to a block of code that corresponds to a \TeX group, then it may be shadowed by a local variable $Y$
\begin{group}
$Y = X$
$Y$
endgroup
\text{⟨arbitrary code making global assignments to $X$⟩}
$X = Y$endgroup.

The inner group is needed to protect $Y$ from being changed.

This is effectively the situation in the blockarray environment, where the outer group is provided by \begin...\end, and the inner group is provided by an assignment to a box register.

• Generating new command names, according to nesting depth. Instead of directly using $X$, the variable can always be indirectly accessed by \csname nesting \endcsname. Here \text{nesting} should expand to a different sequence of tokens for each nested scope in which $X$ is used. \text{nesting} might be altered by a local assignment, or sometimes need to be globally incremented at the start of the scope, and globally decremented at the end.

• Maintaining a stack of previous values. Corresponding to a macro, $X$, is a macro $X\text{stack}$ which consists of a list of the values of $X$ in all outer environments. When the local scope ends, this stack is popped, and the top value (which was the value of $X$ before the environment) is globally assigned to $X$.

The first method has the advantage that the variable is normally accessed within the environment, and the code to restore previous values is trivial. The main memory usage is in the save-stack, \TeX's normal place for saving local values.

Shadowing can only be used when the environment corresponds to a \TeX group. The block environment does not, \text{\end{block}} is not in the scope of any local assignments made by \text{\begin{block}}.

The second method, has the advantage that, once the access functions are defined, it is easy to declare new local variables, however unless you keep track of what has been produced, these variables will continue to take up memory space, even after the environment has ended. blockarray at the moment does not do much clearing up, so after a blockarray there are typically five macros per column per block (u-part, v-part, left right and 'mid' delimiters) left taking up space. Not to mention macros containing the texts of any non-aligned entries.

An extra \text{‘.’} will locally be added to \text{\BA@nesting} as each blockarray is entered, this is used as described above.

43 \text{\def\BA@nesting{}}

These two macros help in accessing macros that are 'local' to the current value of \text{\BA@nesting}.

44 \text{\def\BA@expafter#1#2\{\}

45 \text{\expandafter\#\csname BA\BA@nesting\#2\endcsname}
These are similar, but for macros which depend on the column and block involved, not just the outer blockarray environment.

The following macros manage a stack as described in the third method above.

2.4 The Block Environment
2.5 Multicolumn

First we have the \texttt{\multicolumn} command to be used as in original \LaTeX.

\begin{verbatim}
\def\BAmulticolumn#1#2#3{\multispan{#1}\global\advance\BA@col#1\relax\edef\BA@nesting{\BA@nesting,}\BA@expafter\def{blocktype}{0}{\BA@defcolumntype{&}##1\BA@parseend{\@latexerr{\string& in multicolumn!}\@ehc\BA@parse\BA@parseend}\global\BA@expafter\def{blank@row}{\crcr}\BA@clear@entry\global\let\BA@l@expr\@empty\global\let\BA@r@expr\@empty\BA@colseptrue\BA@parse#2\BA@parseend}\BA@strut\BA@col@use{u}\ignorespaces#3\BA@vpart}
\end{verbatim}

Now something more interesting, a \texttt{\BAmulticolumn} column specification!

\begin{verbatim}
\def\BA@make@mc#1{\count@#1\relax\BA@make@mcX\edef\BA@mc@hash{\noexpand\BA@parse>{\BA@mc@spans}\noexpand\BA@MC@restore@hash\BA@mc@amps\noexpand\BA@MC@switch@amp}}
\def\BA@make@mcX{\ifnum\count@=\@ne\Ba@multicolumn@spans\else\BA@multicolumn@empty\global\let\BA@l@expr\@empty\global\let\BA@r@expr\@empty\BA@clear@entry\global\let\BA@l@expr\@empty\global\let\BA@r@expr\@empty\BA@colseptrue\BA@parse#2\BA@parseend\BA@strut\BA@col@use{u}\ignorespaces#3\BA@vpart}
\end{verbatim}
\def\BA@mc@spans{\null}  
\let\BA@mc@amps=\empty  
\else  
\advance\count@ by\m@ne  
\BA@make@mcX  
\GC@add@to@end\BA@mc@spans{\span}  
\GC@add@to@end\BA@mc@amps{&@{}{}}  
\fi

\BAmultirow

First as a command.
\long\def\BAmultirow#1{\kern#1\relax  
\global\BA@quickfalse  
\BA@col@expafter\gdef{mid}}

Then as a column specification. (The actual \BA@newcolumn comes later)
\def\BA@mrow@bslash#1{\kern#1\relax  
\global\BA@quickfalse  
\iffalse{\else\let\\cr=\textbackslash\fi}\iffalse\fi\iffalse\fi\fi  
\BA@mrow}
\long\def\BA@mrow#1\BA@vpart{\BA@col@expafter\GC@add@to@end{mid}{\endgraf#1}  
\BA@vpart}

\BAnoalign
\def\BAnoalign%\crcr  
\noalign{\ifnum0=’\fi}  
\global\BA@quickfalse  
\penalty\the\BA@row  
\@ifstar{\penalty\GC@four\BA@noalign}{}{\penalty\BA@use{blocktype}\penalty\thr@@\BA@noalign}
\long\def\BA@noalign#1{}{\long\BA@expafter\gdef{noalign\the\BA@row}{#1}{}{\fi}}

\BAnoalign

The following code is taken directly from \texttt{array.sty}, apart from some name changes. It is very similar to the version in \texttt{latex.tex}. Making \textbackslash{} into a macro causes problems when you want the entry to be taken as a macro argument. One possibility is to \texttt{\let} \textbackslash{} be \texttt{span}, and then have the \texttt{⟨u-part⟩} of a final column parse the optional argument. There is still a problem with \texttt{\end{...}}. Note that at the moment this style assumes that \textbackslash{} is used at the end of all lines except the last, even before \texttt{\begin{block}} or \texttt{\end{block}}, this allows spacing to be
specified, and also approximates to the truth about what is actually happening. The idea of making it easier to allow entries to be taken as arguments may be a non-starter if \& is allowed to become a ‘short-ref’ (ie \texttt{active}) character.

\begin{verbatim}
158 \def\BA@cr{(\ifnum 0=')}\fi
159 \def\BA@xcr \BA@xcr\cr
160 \def\BA@argcr {(\ifnum 0='\fi)\cr}
161 \def\BA@argcr[1]{(\ifnum 0='\fi)\cr \ifdim #1>\z@ \BA@xargcr{#1}\else \BA@yargcr{#1}\fi}
162 \def\BA@argcr[1](){\ifnum 0='\fi}\cr}
163 \def\BAargcr[1](){\ifnum 0='\fi}\cr}
164 \def\BAargcr[1](){\unskip}
165 \ifnum \ht\@arstrutbox >\dp\@arstrutbox \vrule \ht\@arstrutbox \dp\@arstrutbox \z@ \cr}
166 \ifnum \ht\@arstrutbox >\dp\@arstrutbox \vrule \ht\@arstrutbox \dp\@arstrutbox \z@ \cr}
167 \def\BA@yargcr[1]{\cr
168 \unskip \noalign{\vskip #1}}
169 \newdimen\BAextrarowheight
170 \newdimen\BAextraheightafterhline
171 \newdimen\BAarrayrulewidth
172 \BAarrayrulewidth\arrayrulewidth
173 \BAarrayrulewidth\arrayrulewidth
174 \newdimen\BAextraruleheight
175 \BAextraruleheight\arrayrulewidth
176 \BAextraruleheight\arrayrulewidth
177 \BAextraruleheight\arrayrulewidth
178 \BAextraruleheight\arrayrulewidth
179 \BAextraruleheight\arrayrulewidth
180 \BAextraruleheight\arrayrulewidth
181 \let\BAstrut\unskip
182 \let\BAstrutA\BAstrut
183 \let\BAstrutB\BAstrut
184 \let\BA@begin\begin
185 \let\BA@end\end
186 \let\BA@begin#1{\expandafter\testGC@x\csname BA@begin#1\endcsname\relax{\BA@@begin{#1}}{\csname BA@begin#1\endcsname}}
187 \let\BA@end#1{\expandafter\testGC@x\csname BA@end#1\endcsname\relax{\BA@@end{#1}}{\csname BA@end#1\endcsname}}
\end{verbatim}

The B form of the strut is an extra high strut to use after a horizontal rule.

\texttt{\begin{block} is supposed to expand to \texttt{\noalign{...}}, but the code for \texttt{\begin} would place non-expandable tokens before the \texttt{\noalign}. Within the \texttt{blockarray} environment, redefine \texttt{\begin} so that if its argument corresponds to a command \texttt{\BA@begin\texttt{[argument]}}, then directly expand that command, otherwise do a normal \texttt{\begin}. A matching change is made to \texttt{\end}.

\begin{verbatim}
181 \let\BA@begin\begin
182 \let\BA@end\end
183 \def\BA@begin#1{\expandafter\testGC@x\csname BA@begin#1\endcsname\relax{\BA@begin#1}\endcsname}}
184 \let\BA@end#1{\expandafter\testGC@x\csname BA@end#1\endcsname\relax{\BA@end#1}\endcsname}}
\end{verbatim}

2.9 Begin and End

\texttt{\begin\texttt{[block]} is supposed to expand to \texttt{\noalign{...}}, but the code for \texttt{\begin} would place non-expandable tokens before the \texttt{\noalign}. Within the \texttt{blockarray} environment, redefine \texttt{\begin} so that if its argument corresponds to a command \texttt{\BA@begin\texttt{[argument]}}, then directly expand that command, otherwise do a normal \texttt{\begin}. A matching change is made to \texttt{\end}.

\begin{verbatim}
181 \let\BA@begin\begin
182 \let\BA@end\end
183 \def\BA@begin#1{\expandafter\testGC@x\csname BA@begin#1\endcsname\relax{\BA@begin#1}\endcsname}}
184 \let\BA@end#1{\expandafter\testGC@x\csname BA@end#1\endcsname\relax{\BA@end#1}\endcsname}}
\end{verbatim}
2.10 The blockarray Environment

Currently I use \everycr this means that every macro that uses \halign that might be used inside a blockarray must locally clear \everycr. The version of array in array.sty does this, but not the one in latex.tex.

The \extrarowheight code from array.sty.

As explained above various registers which are ‘local’ to blockarray are always accessed globally, and so must be shadowed by local copies, so that the values can be restored at the end.

If we are using tablenotes, shadow the footnote counter (or possibly mpfootnote), and set up the print style for the table notes.
Locally increase \textit{BA@nesting} so that macros accessed by ‘the second method’ will be local to this environment.

Now start up the code for this block

There had to be a \texttt{\halign} somewhere, and here it is!

Currently I am using a ‘repeating preamble’ because it was easier, but I think that I should modify the column types for \texttt{blockarray} (not \texttt{block}) so that they construct a preamble with the right number of columns. This would give better error checking, and would give the possibility of modifying the tabskip glue.

The code to place the footnote texts at the foot of the table, each footnote starting on a new line. This will only be activated if \texttt{BA@tablenustrue}.

\begin{verbatim}
\def\BA@expft[#1]#2{\noindent\strut\ifodd0#11{\edef\@thefnmark{\BA@fnsymbol{#1}}\@makefnmark}\else{#1}\fi\ #2\unskip\strut\par}
\end{verbatim}
After a \texttt{\textbackslash Aparfootnotes} declaration, the table notes will be set in a single paragraph, with a good chance of line breaks occurring at the start of a footnote.

\begin{verbatim}
\def\BAparfootnotes{\def\BA@expft[#1][#2]{\nointent\strut\ifodd0##11{\edef\@thefnmark{\BA@fnsymbol{##1}}\@makefnmark}\else{##1~}\fi\hskip \z@ plus 3em \penalty\z@\hskip 2em plus -2.5em minus .5em}}\def\endblockarray{\ifnoalign\or\BAmultirow\def\BA@quick\BA@quick\end\BA@work\BA@backup\leavevmode\BA@finalposition\BA@firstbox}
\end{verbatim}

At this point, if no delimiters, \texttt{\textbackslash Anoalign}, or \texttt{\textbackslash Amultirow} have been used, just finish here, this makes \texttt{blockarray} was just about as efficient as \texttt{array} If no fancy tricks have been used.

Now we restore the values that have been ‘shadowed’ by versions that are local to this environment.

\begin{verbatim}
\global\BA@block@cnt=\BA@block@cnt@shadow\global\BA@col=\BA@col@shadow\global\BA@row=\BA@row@shadow\global\setbox\BA@final@box=\box\BA@final@box@shadow\global\let\BA@delrow=\BA@delrow@shadow\global\let\testBA@quick\testBA@quick@shadow
\end{verbatim}

If tablenotes are being used, reset the shadowed list of footnotes. Otherwise execute the list now, to pass the footnotes on to the outer environment, or the current page.

\begin{verbatim}
\testBA@tablenotes\global\BA@mpftn=\BA@ftn@shadow\global\BA@ftn=\expandafter{\the\BA@ftnx@shadow}\global\BA@ftn\expandafter{\expandafter}\the\BA@ftn}
\end{verbatim}

Here is the ‘quick ending’: just position the box as specified by [tcb], possibly adding footnotes.

\begin{verbatim}
\def\BA@quick\end{\crcr\egroup% end of halign\ifnum0='{\fi}% end of \BA@firstbox\begin{verbatim}
\end{verbatim}
\end{verbatim}

If any delimiters or noaligns have been used, we must take apart the table built in \texttt{\BA@firstbox}, and reassemble it. This is done by removing the rows one by one, starting with the last row, using \texttt{\lastbox}.

\begin{verbatim}
\def\BA@work\BA@backup{\BA@use{blank@row}%\begin{verbatim}
\end{verbatim}
\end{verbatim}

18
If tablenotes are being used, package the table in a box with the notes.

\newdimen\BAfootskip
\BAfootskip=1em
\def\BA@position@c#1{%testBAtablenotes
  \let\footnotetext\BA@expft
  \hsize\wd#1\@parboxrestore\footnotesize{}\vcenter{%testBAtablenotes
    \unvbox#1%testBAtablenotes
    \vskip\BAfootskip\the\BA@ftn}{}}
\def\BA@position@t#1{%testBAtablenotes
  \let\footnotetext\BA@expft
  \hsize\wd#1\@parboxrestore\footnotesize{}%testBAtablenotes
  \unvbox#1%testBAtablenotes
  \vskip\BAfootskip\the\BA@ftn}{}}
\def\BA@position@b#1{%testBAtablenotes
  \let\footnotetext\BA@expft
  \hsize\wd#1\@parboxrestore\footnotesize{}%testBAtablenotes
  \unvbox#1%testBAtablenotes
  \vskip\BAfootskip\the\BA@ftn}{}}
\def\BA@position@st#1{%testBAtablenotes
  \let\footnotetext\BA@expft
  \hsize\wd#1\@parboxrestore\footnotesize{}%testBAtablenotes
  \unvbox#1%testBAtablenotes
  \vskip\BAfootskip\the\BA@ftn}{}}
2.11 Fitting the Parts Together

Get the widths of each column. It also faithfully copies the tabskip glue, even though currently this is always 0pt. The width of the column is put into `\BA@delrow` as the argument to the (unexpanded) call to `\BA@lr`.

```latex
\def\BA@getwidths{%
   \setbox\@tempboxa=\hbox{\unhbox\@tempboxa
   \xdef\BA@delrow{\hskip\the\lastskip}\unskip
   \let\BA@lr\relax
   \loop
   \setbox\BA@tempbox@a=\lastbox
   \ifnum\x=-\hskip\z@=\lastskip\unskip
   \ifhbox\BA@tempbox@a
   \xdef\BA@delrow{%
   \hskip\the\z@\BA@lr{\the\wd\BA@tempbox@a}\BA@delrow}%
   \repeat}
```

The main mechanism by which `blkarray` leaves information to be used 'on the way back' is to leave groups of penalties in the main box. The last penalty in each group (the first to be seen on the way back) is a code penalty, it has the following meanings:

\begin{itemize}
  \item \texttt{\penalty 1} --- \texttt{\begin{block}}
  \item \texttt{\penalty 2} --- \texttt{\end{block}}
  \item \texttt{\penalty 3} --- \texttt{\BA@noalign}
  \item \texttt{\penalty 4} --- \texttt{\BA@noalign*}
  \item \texttt{\penalty 5} --- \texttt{\begin{blockarray}}
\end{itemize}

Note that the `block*` environment does not produce any penalties, this environment is just as efficient as `\multicolumn`, and does not require the second phase, coming back via `\lastbox`.

Above the code penalty may be other penalties, depending on the code, typically these have the values of the blocktype for the block, or the row number.

```latex
\def\BA@check@pen{%
   \count@=\lastpenalty\unpenalty
   \ifcase\count@
   Grumble Grumble. \texttt{\lastpenalty} should be \texttt{void} if the previous thing was not a penalty, and there should be an `\ifvoid\lastpenalty` or something equivalent to test for this. If the user manages to get a `\penalty 0` into the main box, it will just have to be discarded. Actually that is not disastrous, but if a rule, mark, special, insert or write gets into that box `\blockarray` will go into an infinite loop. Every class of \TeX object should have a `\last`... so that boxes may be taken apart and
```

20
reconstructed by special styles like this. \TeX\ of course is frozen, so these missing
features will never be added (while the system is called \TeX).

\begin{verbatim}
358 ⟨∗tracing⟩
359 \ifnum\BAtracing>\tw@\typeout{0-???}\fi
360 ⟨/tracing⟩
361 \BA@get@row
362 \or
363 ⟨∗tracing⟩
364 \ifnum\BAtracing>\tw@\typeout{1-block}\fi
365 ⟨/tracing⟩
366 \BA@expafter\xdef{blocktype}{\the\lastpenalty}\unpenalty
367 \ifnum\lastpenalty=\tw@
368 ⟨∗tracing⟩
369 \ifnum\BAtracing>\tw@\typeout{discarding 2-endblock}\fi
370 ⟨/tracing⟩
371 \unpenalty\unpenalty\fi
372 \BA@place
373 \or
374 ⟨∗tracing⟩
375 \ifnum\BAtracing>\tw@\typeout{2-endblock}\fi
376 ⟨/tracing⟩
377 \BA@expafter\xdef{blocktype}{\the\lastpenalty}\unpenalty
378 \BA@place
379 \or
380 ⟨∗tracing⟩
381 \ifnum\BAtracing>\tw@\typeout{3-BA\noalign}\fi
382 ⟨/tracing⟩
383 \BA@expafter\xdef{blocktype}{\the\lastpenalty}\unpenalty
384 \BA@place
385 \count@=\lastpenalty\unpenalty
386 \global\setbox\BA@final@box=\vbox{%
387 \hspace=\wd\BA@final@box\parboxrestore
388 \vrule \@height \ht\@arstrutbox \@width \z@\n
389 \BA@use{noalign\the\count@}
390 \vrule \@width \z@ \@depth \dp \@arstrutbox
391 \endgraf\unvbox\BA@final@box%
392 \or
393 ⟨∗tracing⟩
394 \ifnum\BAtracing>\tw@\typeout{4-BA\noalign*}\fi
395 ⟨/tracing⟩
396 \count@=\lastpenalty\unpenalty
397 \setbox\BA@block@box=\vbox{%
398 \hspace=\wd\BA@final@box \parboxrestore
399 \vrule \@height \ht\@arstrutbox \@width \z@\n
400 \BA@use{noalign\the\count@}
401 \vrule \@width \z@ \@depth \dp \@arstrutbox
402 \endgraf\unvbox\BA@block@box%
403 \or
404 ⟨∗tracing⟩
\end{verbatim}

21
Move a row of the table from the \BA@first@box into the block that is being constructed in \BA@block@box.

\def\BA@get@row{% 
\skip@=\lastskip\unskip 
\advance\skip@=\lastskip\unskip 
\setbox\@tempboxa=\lastbox 
\setbox\BA@block@box=\vbox{% 
\box\@tempboxa 
\vskip\skip@ 
\unvbox\BA@block@box}

Place the block that has been constructed in \BA@block@box, together with any delimiters, or spanning entries which have been assembled into \BA@delrow, into the final table, which is being constructed in \BA@final@box.

\def\BA@place{% 
\global\setbox\BA@final@box=\vbox{\hbox{% 
\m@th\nulldelimiterspace=\z@ 
\divide\dimen\z@ by \ht\BA@block@box 
\dimen2=\dimen\z@ 
\advance\dimen\z@ by -\fontdimen22 \textfont2 
\global\BA@col=\z@ 
\BA@delrow 
\kern-\wd\BA@block@box 
\ht\BA@block@box=\dimen\z@ \dp\BA@block@box=\dimen\tw@ 
\box\BA@block@box 
\unvbox\BA@final@box}}

Place the delimiters or spanning entries in position for one column of the current block.

\def\BA@l#1{% 
\global\advance\BA@col=\one \relax 
\BA@col@use{left}% 
\BA@col@expafter{\ifx[mid]\@empty 
\kern#1 
\else 
\hbox to #1% 
\setbox\BA@tempboxa a
2.12 Parsing the Column Specifications

A token \texttt{x} in the column specification, is interpreted by \texttt{\textbackslash \textbackslash parse} as \texttt{\textbackslash BA@<x>}. This command is then expanded, which may take further tokens as arguments. The expansion of \texttt{\textbackslash BA@<x>} is supposed to end with a call to \texttt{\textbackslash parse} which will convert the token following any arguments to a control sequence name. The process is terminated by the token \texttt{\textbackslash parseend} as the corresponding command \texttt{\textbackslash \textbackslash parse\textbackslash parseend} does some ‘finishing off’, but does not call \texttt{\textbackslash parse}.

There are two commands to help in defining column types.

\texttt{\textbackslash defcolumntype} This takes its parameter specification in the primitive \texttt{\def} syntax, and allows the replacement text to be anything.

\texttt{\newcolumntype} This takes its parameter specification in \LaTeX{}’s \texttt{\newcommand} syntax, and applies \texttt{\textbackslash parse} to the \texttt{front} of the replacement text. This is intended for users to define their own column types in terms of primitive column types, rather than in terms of arbitrary \TeX{} expressions.

The preamble argument build up various macros as follows:

Each entry in the \texttt{\align} preamble is of the form

\texttt{\upart\vpart}\texttt{\upart} increments \texttt{\col}, and then expands \texttt{\col\use{u}}, similarly \texttt{\vpart} expands \texttt{\col\use{v}}.

\texttt{\col\use{u}} is a macro considered local to the current block and column, it is always accessed via \texttt{\col\use{u}} or \texttt{\col\expafter}. So the preamble entries must result in \texttt{\col\use{u}} and \texttt{\col\use{v}} being defined to enter any text specified in \texttt{\&}-expressions, the inter-column space (in the \TeX{} tradition, not \texttt{\tabskip}), and any declarations in \texttt{\&} and \texttt{<} expressions. Delimiters are not added to these macros as they correspond to the whole block, they are left in the macros \texttt{\col\use{left}} and \texttt{\col\use{right}} spanning entries from \texttt{\multirow} are considered like delimiters, and left in \texttt{\col\use{mid}}.
If the test `\textBA@upart` is true, then the ⟨u-part⟩ is being built. This consists of three different sections.

1) The left inter-column text, declared in !- and @-expressions
2) The left inter-column skip.
3) Any declarations specified in > expressions.

Suppose X is a column type, which may itself be defined in terms of other column types, then the (equivalent) specifications:

\@{aaa}>{\foo}X and >{\foo}@{aaa}X

must result in aaa, surrounded by \bgroup...\egroup or $...$, being prepended to the front of the u-part, and \foo being appended to the end, so that it is the innermost declaration to be applied to the entries in that column.

In order to achieve this, > expressions are directly added to the u-part, using \GC@add@to@front, @- and !- expressions (and rules from 1) are added to a scratch macro, \BA@l@expr, using \GC@add@to@end.

When the next # column specifier is reached, the \BA@l@expr is added to the front of the u-part, It is separated from the >-expressions by a ⟨hskip⟩ unless an @-expression has occurred, either while building the current u-part, or the previous v-part.

Building the v-part is similar.

This procedure has certain consequences,

- \@{a}@{b} is equivalent to \@{ab}, or more correctly \@{{a}{b}}.
- >{a}>{b} is equivalent to >{b}{a}.
- If any @-expression occurs between two columns, all !-expressions between those columns will be treated identically to @-expressions. This differs from \textarray.sty where two !-expressions are separated by the same skip as the rules specified by \|.
- If any rule | occurs, then a following rule will be preceded by the doubleruleskip, unless a @ or !-expression comes between them. In particular |&| specifies a double rule, which looks the same as |\|, but \textmulticolumn commands can be used to remove one or other of the rules for certain entries.

Create a macro name from a column specifier.

\def\BA@stringafter#1#2{\expandafter#1\csname BA@<$\string$#2>$\endcsname}

Execute the specifier, or discard an unknown one, and try the next token.

\def\BA@parse#1{%
  \@latexerr {Unknown column type, \string#1}\@ehc\BA@parse%
  \csname BA@<$\string$#1>$\endcsname}

\def for column types.

\def\BA@defcolumntype#1{%
  \BA@stringafter\def#1
\newcommand for column types.¹

\newif\ifBA@colsep
\newif\ifBA@rulesep

These \ifs will be true if their associated skips are to be added in the current column.

\newif\ifBA@colsep
\newif\ifBA@rulesep

This is true if we do not need to come back up the array.

This will be true if building the \langle u-part \rangle, and false if building the \langle v-part \rangle.

\GC@newtest{BA@quick}
\GC@newtest{BA@upart}

2.13 ‘Internal’ Column-Type Definitions

> expressions:
If we are building the v-part, add a &, and try again, so that \( c \{\{a\}\} \{\{b\}\} c \) is equivalent to \( c \{\{a\} \& \} \{\{b\}\} c \).
Otherwise add the expression to the front of the u-part, i.e., the list being built in \( \text{\BA@col@use(u)} \). Note that no grouping is added so that the scope of any declaration includes the column entry.

\BA@defcolumntype{>}{#1}{%
\testBA@upart
{\BA@col@expafter\GC@add@to@front{u}{#1}%
\BA@parse}%
{\BA@parse &>{#1}}}

Left delimiters:
Again add a & if required, otherwise just save the delimiter and label as the first two arguments of \( \text{\BA@left@del} \) in the macro \( \text{\BA@col@use(left)} \).

\BA@defcolumntype{\Left}{#1#2}{%
\testBA@upart
{\global\BA@quickfalse
\BA@col@expafter\gdef{left}{\BA@leftdel{#1}{#2}}\BA@parse}%
{\BA@parse &\Left{#1}{#2}}}

Right delimiters: As for Left.

\BA@defcolumntype{\Right}{#1#2}{%
\testBA@upart
{\global\BA@quickfalse
\BA@col@expafter\gdef{right}{\BA@rightdel{#1}{#2}}\BA@parse}%
{\BA@parse &\Right{#1}{#2}}}

The end of each column specification is terminated by & either by the user explicitly entering &, or one being added by one of the other rewrites.

¹Currently does not check that the type is new.
If we are still in the u-part, finish it off with #.
Otherwise add another column to the blank row, advance the column counter by one. Finally reset the variables \BA@use{u|v|left|mid|right}.

\BA@defcolumntype{(}\{%
\testBA@upart{\BA@parse ##&}%
\BA@expafter\xdef{blank@row}{\BA@use{blank@row}\omit&}%
\global\advance\BA@col\@one
\BA@clear@entry
\BA@parse}\%

# Add a & if required. 
Otherwise make the u-part of the current column, and the v-part of the previous one.

\BA@defcolumntype{#}(\%{\%
\testBA@upart{\%}

Add the intercolumn skips unless a \& expression has occured since the last # entry.

\ifBA@colsep
\GC@add@to@front\BA@r@expr{\BA@edollar\hskip\BA@colsep}\
\GC@add@to@end\BA@l@expr{\hskip\BA@colsep\BA@bdollar}\
\else
\GC@add@to@front\BA@r@expr{\BA@edollar}\
\GC@add@to@end\BA@l@expr{\BA@bdollar}\
\fi

Go back to the previous column. Add \BA@r@expr to the end of \BA@col@use{v}.

\global\advance\BA@col\m@ne
\BA@col@expafter\GC@add@to@end{v\expandafter}\expandafter{\BA@r@expr}\

Add the total width of any \& expressions as the third argument in the right delimiter macro, this will be used to move the delimiters past any inter-column material.

\BA@add@rskip

Repeat for the u-part, and left delimiter of the current column.

\global\advance\BA@col\m@ne
\BA@col@expafter\GC@add@to@front{u\expandafter}\expandafter{\BA@l@expr}\

Clear these scratch macros ready for the next column.

\global\let\BA@l@expr\@empty\global\let\BA@r@expr\@empty

Reset these tests and ifs, ready for the next inter-column material.

\BA@upartfalse
\BA@rulesepfalse
\BA@colseptrue

26
Finally look at the next specifier.

\[ \texttt{\textbackslash parse}\]$\%$
\[ \texttt{\textbackslash parse \&##}}\]

< Just like >.

\[ \texttt{\textbackslash defcolumntype\{<\#1\%\texttt{\textbackslash test\textbackslash upart}}\]
\[ \texttt{\textbackslash parse \{\#1\}}\%$
\[ \texttt{\textbackslash col@expafter\texttt{\textbackslash GC@add@to@front\{\texttt{\textbackslash v}\#1\}}\texttt{\textbackslash parse}}\]

BA version of \vline.

\[ \texttt{\textbackslash def\textbackslash BA@vline\{\textbackslash vrule \textbackslash \textbackslash @width \textbackslash \textbackslash \textbackslash \textbackslash arrayrulewidth}}\]

| like !, except that a \hskip\BA@doublerulesep is added for consecutive pairs.

\[ \texttt{\textbackslash BA@defcolumntype\{\}}\%$
\[ \texttt{\textbackslash test\textbackslash upart}}\]
\[ \texttt{\if\textbackslash BA@rulesep\texttt{\else}}\%
\[ \texttt{\GC@add@to@end\textbackslash BA@l@expr\{\hskip\BA@doublerulesep\textbackslash BA@vline\}}\%
\[ \texttt{\fi}}}\%$
\[ \texttt{\if\textbackslash BA@rulesep\texttt{\else}}\%
\[ \texttt{\GC@add@to@end\textbackslash BA@r@expr\{\textbackslash BA@vline\}}\%
\[ \texttt{\fi}}}\%$
\[ \texttt{\BA@ruleseptrue}\%
\[ \texttt{\textbackslash parse}}\%

\[ @\] identical to !, but set \BA@colsepfalse.

\[ \texttt{\textbackslash BA@defcolumntype\{\}}\#1\%$
\[ \texttt{\textbackslash test\textbackslash upart}}\]
\[ \texttt{\if\textbackslash BA@rulesep\texttt{\else}}\%
\[ \texttt{\GC@add@to@end\textbackslash BA@l@expr\{\textbackslash BA@dollar\#1\textbackslash BA@edollar\}}\%
\[ \texttt{\fi}}}\%$
\[ \texttt{\if\textbackslash BA@rulesep\texttt{\else}}\%
\[ \texttt{\GC@add@to@end\textbackslash BA@r@expr\{\textbackslash BA@dollar\#1\textbackslash BA@edollar\}}\%
\[ \texttt{\fi}}}\%$
\[ \texttt{\BA@ruleseptrue}\%
\[ \texttt{\textbackslash parse}}\%

! Just save the expression, and make \BA@rulesepfalse so that the next | is not preceded by a skip.

\[ \texttt{\textbackslash BA@defcolumntype\{\}}\#1\%$
\[ \texttt{\textbackslash test\textbackslash upart}}\]
\[ \texttt{\if\textbackslash BA@rulesep\texttt{\else}}\%
\[ \texttt{\GC@add@to@end\textbackslash BA@l@expr\{\textbackslash BA@dollar\#1\textbackslash BA@edollar\}}\%
\[ \texttt{\fi}}}\%$
\[ \texttt{\if\textbackslash BA@rulesep\texttt{\else}}\%
\[ \texttt{\GC@add@to@end\textbackslash BA@r@expr\{\textbackslash BA@dollar\#1\textbackslash BA@edollar\}}\%
\[ \texttt{\fi}}}\%$
\[ \texttt{\BA@rulesepfalse}\%
\[ \texttt{\textbackslash parse}}\%

*: *\{3\}\{xyz\} just produces xyz*\{2\}\{xyz\} which is then re-parsed.
\edef\@tempa##1{\noexpand\BA@parse##1*{\the\count@}{##1}}%
\else
\edef\@tempa##1{\BA@parse}%
\fi
\@tempa{#2}}

\BA@parseend this is added to the end of the users preamble, it acts like a
cross between \# and \&. It terminates the preamble building as it does not call
\BA@parse.
\BA@defcolumntype{\BA@parseend}{%
\testBA@upart
{\BA@parse ##\BA@parseend}%
{\
\ifBA@colsep
\GC@add@to@front\BA@r@expr{\BA@edollar\hskip\BA@colsep}%
\else
\GC@add@to@front\BA@r@expr{\BA@edollar}%
\fi
\BA@expafter\xdef{blank@row}{\BA@use{blank@row}\omit\cr}%
\BA@add@rskip
\BA@col@expafter\GC@add@to@end{v\expandafter}\expandafter
{\BA@r@expr}}}

Like array.sty
\def\BA@startpbox#1{\bgroup
\hsize #1 \@arrayparboxrestore
\vrule \@height \ht\@arstrutbox \@width \z@}
\def\BA@endpbox{\vrule \@width \z@ \@depth \dp \@arstrutbox \egroup}
\def\BA@endpbox{\vrule \@width \z@ \@depth \dp \@arstrutbox \egroup}

Save the \& and \# macros, so they can be restored after a multicolumn, which
redefines them.
\BA@stringafter{\let\expandafter\BA@save@amp}{&}
\BA@stringafter{\let\expandafter\BA@save@hash}{#}

A column specification of \BAmulticolumn{3}{c} is re-written to:
c\BA@MC@end
28
The special definition of \& while parsing the multicolumn argument.

Putting it all together!

As explained above, in order to position the delimiters on the way back we need the widths of the inter-column texts.

2.14 User Level Column-Type Definitions

\def\newcolumntype{c} {>{\hfil}<{\hfil}}
\def\newcolumntype{l} {>{}<{\hfil}}
\def\newcolumntype{r} {>{\hfil}<{}}
\def\newcolumntype[p]{1}{>{\vtop\startpbox{#1}}c<{\endpbox}}
\def\newcolumntype[m]{1}{>{$\vcenter\startpbox{#1}}c<{$\endpbox$}}
\def\newcolumntype[b]{1}{>{\vbox\startpbox{#1}}c<{\endpbox}}
\def\newcolumntype{(}{\Left{}{(}}
\def\newcolumntype{)}{\Right{)}{}}
\def\newcolumntype[}{\Left{}{}}
\def\newcolumntype]{}{\Right{}{}}
\def\newcolumntype[\BAenum]{% !{% 
\def\protect\noexpand\protect\noexpand}%
2.15 Footnotes

This test is true if footnote texts are to be displayed at the end of the table.

Inside the alignment just save up the footnote text in a token register.

2.16 Hline and Hhline

The standard \hline command would work fine ‘on the way down’ but on the way back it throws me into an infinite loop as there is no \lastrule to move the rule into the final box. I could just make \hline leave a code penalty, and put in the rule on the way back, but this would mean that every array with an \hline needs to be taken apart. I hope to make ‘most’ arrays be possible without comming back up the array via \lastbox. I could do something with \leaders which are removable, but for now, I just make \hline and \hline\hline just call \hhline with the appropriate argument. The \hhline from hhline.sty does work, but needs extra options to deal with \& etc, but here is a re-implementation, more in the spirit of this style.

First set up the boxes used in \leaders.
denotes a broken vertical rule, as in hhline.sty. If the double dots : : : : currently produced by * turn out to be useful, it might be better to use : for them, and something else, perhaps ! for this feature.