```
..\tenrm t
..\tenrm h
..\hbox(0.0+0.0)x0.00002, glue set - 0.59999fil,
     shifted -6.0
...\glue 0.0 plus 1.0fil minus 1.0fil
...\hbox(0.0+0.0)x0.0
....\tensy 7
...\kern 1.2
...\glue 0.0 plus 1.0fil minus 1.0fil
..\tenrm e
........
........
..\tenrm g
..\tenrm .
..\penalty 10000
..\glue(\parfillskip) 0.0 plus 1.0fil
..\glue(\rightskip) 0.0
.\glue 0.0 plus 1.0fill

! OK (see the transcript file).
<output> {\showbox 255
           \shipout \box 255 \advancepageno }
\break ->\penalty -\@M

1.12 \vfill\eject

?
```

A summary and a wish

The methods described here have limitations and disadvantages, so they cannot be used in every situation. Method 2 still has a few unsolved problems. As a result, the macros described here cannot be canned and used 'as is'. They should be carefully studied and understood, so that they could be applied to practical problems. This means that they are beyond the grasp of beginners but, because of their power, they may provide the necessary incentive to many beginners to become full fledged wizards.

It would be so much easier to solve the three problems discussed here if the `\lastbox` command could recognize characters of text, or if a new command, `\lastchar`, were available for this purpose. This is a private wish that I hope will be shared by readers.

Finally, I would like to thank the many people who have responded to the original OTR articles of 1990. I would like to think that I was able to help some of them, and I know that their comments, questions, and criticism have helped me become more proficient in this fascinating field of OTR techniques.

References

1. Salomon, D., *Output Routines: Examples and Techniques. Part II*, *TUGboat* 11(2), pp. 212–236, June 1990.
2. Haralambous, Y., Private communication.
3. Graham, R. L., et al., *Concrete Mathematics*, Addison-Wesley, 1989.
4. Bell, E. T., *Men of Mathematics*, Simon and Schuster, 1937.
5. Knuth, D. E., *Computers and Typesetting*, vol. E, Addison-Wesley, 1986.

◇ David Salomon
California State University, Northridge
Computer Science Department
Northridge, CA 91330-8281
dxs@secs.csun.edu

Verbatim Copying and Listing

David Salomon

A general note: Square brackets are used throughout this article to refer to *The TEXbook*. Thus [39] refers to page 39. Also, the logo OTR stands for 'output routine', and MVL, for 'Main Vertical List'.

Introduction

Methods are developed, and macros listed, to solve the following two problems. Verbatim copying is the problem of writing a token string verbatim on a file, then executing it. Verbatim listing involves typesetting a token string verbatim, in either horizontal or vertical mode.

We start with a short review of `\edef`. In '`\edef\abc{\xyz \kern1em}`', the control sequence `\xyz` is expanded immediately (at the time `\abc` is defined), but the `\kern` command is only executed later (when `\abc` is expanded).

The same thing happens when `\abc` is defined by means of `\def`, and is then written on a file. Thus '`\write\aux{\abc}`' writes the replacement text that would have been created by `\edef\abc{...}`.

Sometimes it is desirable to write the name of a control sequence on a file, rather than its expansion. This can be done either by '`\write\aux{\noexpand\abc}`' or, similarly, by '`\write\aux{\string\abc}`'. The former form

writes a space following \abc, while the latter one does not.

Verbatim copying

With this in mind we now consider the following problem: given an arbitrary string, containing text, control sequences, active characters, and special characters (such as #%{}), first write it on a file without expansion (verbatim), then expand it.

Before delving into the details, here are some examples that show that this problem is practical:

1. When writing a textbook with exercises and answers, the author would like to be able to say:

```
\exercise...
\answer...
...
\endanswer
```

and have the answer written verbatim on a file. The file can later be input, to typeset all the answers in an appendix. However, while the book is being written, the author may also want, for proofreading purposes, to typeset the answer right following the exercise. Note that an answer may contain many control sequences, and may be long.

2. When writing a book on TEX, the author would like to have an active character (say '^'), such that '^{\baselineskip}=24pt' would write \baselineskip on a file (perhaps with the page number, for later preparation of an index) *and also* execute '\baselineskip=24pt'.

We develop two approaches to this problem. The first one uses catcode changes to suppress the special meanings of certain characters before the string is read by TEX. It is then easy to read the string and write it verbatim on a file. However, in order to also expand the string, all characters should have their normal catcodes. This is done by writing the string on another file and reading it back immediately. This way, the string is parsed into tokens that get their normal catcodes, and can later be expanded.

In the second approach no catcodes are modified; the string is input as usual and tokens created. The string is then scanned, token by token, to identify the control sequence tokens. Expansion is avoided either by placing a \noexpand in front of each control sequence, or by temporarily redefining each control sequence as \relax (which is non-expandable). The string can then be written on a file with nothing expanded. Following which, all control sequences get back their original meanings, and the string can be expanded in the usual way.

My usual disclaimer applies heavily to this material and is therefore repeated: the macros presented here are simple. Each has its limitations, and can be used for certain applications only. The macros should therefore not be copied and used verbatim. They should be carefully studied and fully understood by the reader, so that they could be modified for specific applications.

Approach 1. We present a number of macros, all based on catcode changes. The first two change the catcodes of all the special characters. The other three change the catcodes of just a few characters. In between the two groups, we illustrate how the macros can be modified to handle a specific problem, namely, writing index items, with page numbers, on a file.

Basic verbatim copying: \VwriteA. To avoid expansion we change the catcodes of the special characters, such as '$', '#' and '\', to 12 (other). This way, the '\' is no longer the escape character, so TEX does not recognize any control sequences, and there is nothing to expand. The catcode changes should, of course, be done locally, in a group. Macro \VwriteA below starts a group, does the catcode changes, and expands \aux. Macro \aux absorbs the argument, does the \write, and closes the group.

```
\def\makeother#1{\catcode`#1=12\relax}
\def\sanitize{\makeother\ \makeother\\%
 \makeother\$\makeother\&%
 \makeother\#\makeother\_%
 \makeother\%\makeother\~\makeother\|}

\newwrite\out
\immediate\openout\out=filename

\def\VwriteA{\begingroup\sanitize\aux}
\def\aux#1{\write\out{#1{\folio}}\endgroup}
```

The following is a representative expansion '\VwriteA{te xt#$%&_~\a\x fin}'. The argument, which seems to belong to \VwriteA, is actually absorbed by \aux. Also, since the actual writing is done in the OTR, it is possible to write the page number on the file, even in braces, as above.

Any active characters that may appear in our strings should, of course, also be sanitized. In the example above the vertical bar '|' was sanitized, since we declare it active and use it for verbatim listings. The braces, on the other hand, were not sanitized, which makes it possible to enclose the argument in braces (but then the argument cannot contain arbitrary braces, only balanced ones).

Verbatim copying with braces. Macro \VwriteB below is a slightly different version that does sanitize the braces. The argument can now contain arbitrary braces, but it must be delimited by something else (the string 'endP' in our case).

```
\def\sanitize{\makeother\ \makeother\\%
 \makeother\$\makeother\&%
 \makeother\#\makeother\_\makeother\%
 \makeother\~\makeother\|%
 \makeother\}\makeother\{}

\newwrite\out
\immediate\openout\out=filename

\def\VwriteB{\begingroup\sanitize\aux}
\def\aux#1endP{\write\out{#1{\folio}}%
 \endgroup}
```

An example. As a practical example, we use \VwriteA to illustrate the creation of a raw index file. The following macros declare the '^' an active character to write index items (with page numbers) on a file. They also use \futurelet to allow silent index items (items that should be written on the file, but should not be typeset).

Typical uses are ^{kerning}, ^[\kern]{} and ^[B.~L.]{User}. Up to two arguments can be specified, and are written on the index file as one string (with the page number). However, only the main argument (in braces) is typeset. The optional argument, in brackets, is silent.

```
\def\Caret{\ifmmode\def\next{^}%
  \else\let\next=\indexT\fi\next}
\catcode`\^=\active \let^=\Caret

\def\indexT{\futurelet\new\macX}
\def\macX{\ifx\new[\let\Next=\inxB%
  \else\let\Next=\inxA\fi
  \begingroup\sanitize\Next}
\def\inxA#1{#1\writeinx{#1}}
\def\inxB[#1]#2{#2\writeinx{#1#2}}
\def\writeinx#1{\write\inx{%
  \string\indexentry{#1}{\folio}}\endgroup}
```

This example is simple and easy to understand, but it is not completely general. The problem is that an item such as '^\TeX' is expanded when the \futurelet sees it (before sanitizing). Therefore, its expansion, rather than its name, is written on the file. When the item is enclosed in braces '^{\TeX}', the \futurelet only sees the '{', so the item is not immediately expanded. After sanitizing, its name is written on the file. However, because of the sanitizing, the name of the item, rather than its expansion, is also typeset in the document. In the case of index items, the user can write the control sequence twice, once outside the '^' to expand it, and once inside the '^' to write its name on the file. Thus '\TeX^[\TeX]{}'.

In general, a way is needed to write the contents of any string, with no expansion, on a file, then expand it. Unfortunately, sanitizing is done by changing catcodes and, once a catcode is assigned to a token, this assignment is permanent [39] and cannot be changed. A solution exists, however, and is developed below.

Verbatim copying with an auxiliary file. Developing the solution is done in three steps. In step 1, a simple macro, \VwriteC, is developed that can write strings on a file without expanding control sequences. Its limitations are: (a) The macro sanitizes certain characters so, after writing the string on a file, it (the string) cannot always be expanded; (b) A multi-line string is written on the file as one line.

In step 2, limitation (a) above is removed. Macro \VwriteD is a generalization of \VwriteC, which does the following: The string is written on the file as before, and is then written on another file which is immediately \input. When the string is read back, all tokens get their normal catcodes, and the string can be expanded as usual.

In step 3, macro \VwriteD is modified, à la \elp below, to scoop up one input line at a time. The result is called \VwriteE. This way, a multi-line string is written on a file line by line. The string can also be long, since only one line need be saved at a time.

Note that\VwriteE is a generalization of macro \VwriteD which, in turn, has been developed from \VwriteC. The reader is advised to carefully follow the development of all three macros, however, since each has different features and involves different ideas that can be used for other problems, not just verbatim copying.

Step 1: Verbatim copying with a toks register. To write a string on a file without expansion, it is placed in a toks register, and then written from the register.

```
\toks0={...string...}
\immediate\write\out{\the\toks0}
```

This works since the control sequence '\the' creates a string of tokens, all of catcode 12, except spaces. Fortunately, '\the' can be applied to a \toks register. Note that this illustrates an advantage of \toks registers over macros, since '\the' cannot be applied to a macro. Trying to say:

```
\def\aux{...string...}
\immediate\write\out{\aux}
```

would expand \aux and all the commands in it. Things such as '\noexpand\aux' or '\string\aux' would simply write the name of the macro, not its contents, on the file.

In practice, the string to be written on a file is the argument of a macro, so the actual code is:

```
\def\aux#1\endP{\toks0=\expandafter{#1}
 \immediate\write\out{\the\toks0}
 ...}
```

This works since \expandafter is a one-step expansion. It expands '#1' into individual tokens, but does not expand the tokens further. Our first version is thus:

```
\newtoks\str \newwrite\out
\immediate\openout\out=filename
\def\VwriteC{\begingroup\sanitize \aux}
\def\aux#1\endP{\str=\expandafter{#1}%
 \immediate\write\out{\the\str}%
 \endgroup}
\def\sanitize{\catcode`\ =12
 \catcode`\%=12 \catcode`\#=12
 \catcode`\{=12 \catcode`\}=12 }
```

Following which, the macro can be expanded by, e.g.: '\VwriteC a$x\le0$\c C\vrule 1\endP' or '\VwriteC X\hskip10pt$\sum$\endP'

Step 2: Verbatim copying with an auxiliary file. The string written on a file cannot, in general, be expanded by our macros, since the catcodes of the characters '#%{}' (and of the space) were changed. Trying to say, e.g.:

```
\def\aux#1\endP{\str=\expandafter{#1}%
 \immediate\write\out{\the\str}%
 \endgroup #1}
```

might produce wrong results, or an error message, if the string contains a '#', an '&', or any braces.

Macro \VwriteD below solves this problem by writing the string on a second file, and reading it back immediately. Upon reading, the string is parsed into tokens that get their normal catcodes. The string can now be expanded. This is a general (albeit slow) solution to the problem. TEX is coerced into scanning the same string twice, the first time with special catcodes, and the second time, under normal conditions.

```
\newtoks\str \newwrite\out \newwrite\Tmp
\def\sanitize{\catcode`\ =12
 \catcode`\%=12 \catcode`\#=12
 \catcode`\{=12 \catcode`\}=12 }
\immediate\openout\out=\jobname.aux
\def\VwriteD{\begingroup\sanitize \aux}
\def\aux#1\endP{\str=\expandafter{#1}%
 \immediate\write\out{\the\str}%
 \immediate\openout\Tmp=\jobname.tmp
 \immediate\write\Tmp{\the\str}%
 \endgroup \immediate\closeout\Tmp
 \input\jobname.tmp }
```

Step 3: Verbatim copying of a multi-line string. We start by developing a macro \elp (short for End Line Parameter), whose single parameter is delimited by the end of the line. It is used to pick up an entire input line at a time. We cannot simply write \def\elp#1^^M{...} since the ^^M would terminate the current line and send TEX looking for the definition {...} on the next line.

So we try to change the catcode of the end-of-line (carriage return) character. Initially we try to change it to 13 (active). The first attempt is '\def\elp#1^^M{\catcode`\^^M=13...}', but this does not work since, when TEX finds the catcode change, it has already scanned and determined what the argument is. We have to change the catcode before \elp is expanded. The definitions \catcode`\^^M=13 \def\elp#1^^M{...} work, but this means that \elp can only be used when the catcode change is in effect (i.e., inside a group).

A similar solution defines \elp in \obeylines mode [352] (in which ^^M is active). Thus '{\obeylines\gdef\elp#1^^M{...}...}'. It has the same disadvantage as above.

A better solution is to define \elp *without a parameter*, change the catcode inside \elp (by means of \obeylines), and then expand another macro, \getpar, that actually picks up the argument. The result is:

```
\def\elp{\begingroup\obeylines\getpar}
{\obeylines
 \gdef\getpar#1
  {`#1'\endgroup}}
```

Macro \elp performs the catcode change, and expands \getpar. Macro \getpar is thus always expanded when TEX is in \obeylines, but \getpar is also defined inside an \obeylines. The fact that its definition is on a separate line means that its parameter, #1, is delimited by an end-of-line. The \endgroup in \getpar terminates the effect of the catcode change.

The next step is to realize that the catcode of ^^M can be changed simply to 12 (other), and there is no need to bother with active characters. Perhaps the best solution is:

```
\def\elp{\begingroup%
 \catcode`\^^M=12 \elpAux}
{\catcode`\^^M=12 \gdef\elpAux#1^^M{%
 \message{`#1'}\endgroup}}
```

The catcode change is localized by means of \begingroup and \endgroup. An auxiliary macro, \elpAux is used to actually pick up the argument. The macro can be used anywhere.

After this introduction, macro \VwriteE is presented. It can read a long, multi-line, argument and write it on a file line by line. The idea is to have macro \aux scoop up one line of the source string as its argument, write it on the file, then expand \VwriteC recursively until a certain string (\endP in our case) is found, that signals the end of the argument.

The main difference between \VwriteE and \VwriteC is the definition of \aux. The version of \aux that's expanded by \VwriteE below is defined in a group where the catcode of ⟨return⟩ is set to 12. It uses the principles of \elp to scoop up one line of the argument, write it on the file, and expand \VwriteE recursively.

```
\newtoks\str \newwrite\out
\def\sanitize{\catcode`\ =12
 \catcode`\%=12 \catcode`\#=12
 \catcode`\{=12 \catcode`\}=12 }
\immediate\openout\out=\jobname.aux

\def\VwriteE{\begingroup\sanitize%
 \catcode`\^^M=12 \aux}

{\catcode`\^^M=12%
 \gdef\aux#1^^M{\def\temp{#1}%
  \ifx\temp\enP%
   \gdef\next{\relax}%
  \else \str=\expandafter{#1}%
   \immediate\write\out{\the\str}%
   \gdef\next{\VwriteE}%
  \fi\endgroup\next}}

\def\enP{\endP}
```

A typical expansion now looks like:

```
Any text...\VwriteE9A{\bf abc} \B
11\halign{#\cr1\cr}\TeX
x$\yy@#%^&_?}\it {\c
\endP
...more text
```

Notes:

1. The \endP must be on a line by itself, and must start on column 1. The user may, of course, change from '\endP' to any other string.
2. Macro \aux is defined when the catcode of ⟨return⟩ is 12. Therefore, every line in the definition of \aux must be delimited by a '%'. Otherwise the end of line would be typeset as \char`015 in the current font. (The table on [367] shows that '015 is the character code of ⟨return⟩.)
3. The temporary macro \next is defined by '\gdef' instead of by '\let', since it is defined inside \aux and \aux is defined inside a group.
4. It seems that steps 2 and 3 can be combined. It is suggested that the reader develop a macro \VwriteF with the combined features of \VwriteD and \VwriteE.
5. The three macros above write the value of the toks register \str on the file. They therefore cannot use a delayed \write, and must use \immediate\write. Trying to say '\write\out{\the\str}' would delay all the write operations to the OTR, where register \str may contain the string from the most recent write, or may even be undefined.

Approach 2. In this approach there are no catcode changes (except that \obeyspaces is used locally, during scanning). The string is input and is parsed into tokens in the normal way. This way, our macros can expand it by simply saying '#1'. The first version, \VwriteM, scans the string of tokens and inserts a \noexpand in front of every control sequence token. The second version, \VwriteN, does the same scanning, and changes the meaning of every control sequence to \relax. The scanning is done with macros \scan and \onestep, which are based on the last example on [219].

Version 1: verbatim copying with \noexpand. Macro \onestep receives the next token in the string, checks to see if it is a control sequence (by comparing its catcode to that of \relax) and, if it is, inserts a \noexpand in front of it. The new string is created, token by token, in the toks register \str. The only step that needs detailed explaining is macro \temp. It is important to understand why this macro is necessary (why not simply say '\immediate\write\out{\the\str}'), and why the use of \edef?

Consider the expansion '\VwriteM{a\TeX}'. When scanning is complete, the toks register \str contains 'a\noexpand \TeX ' (including the spaces). Now '\immediate\write\out{\the\str}' would write that string (including the \noexpand) on the file, as in approach 1 above. Defining \temp by means of \def would make '\the\str' the replacement text of \temp, so the command

'\immediate\write\out{\temp}' expands \temp and would be identical to writing '\the\str'. The \edef, however, creates 'a\noexpand \TeX ' as the replacement text of \temp. During the write operation \temp is expanded, which is when the \noexpand does its job and prevents the expansion of \TeX.

```
\newwrite\out \newtoks\str
\immediate\openout\out=filename

\def\VwriteM#1{'#1'{\str={}%
 \obeyspaces\scan#1\end
\edef\temp{\the\str}
\immediate\write\out{\temp}}}

\def\scan#1#2\end{\def\aux{#1}%
 \ifx\aux\empty
 \else
  \def\aux{#2}%
  \onestep{#1}%
  \ifx\aux\empty
  \else
   \scan#2\end
 \fi\fi}

\def\onestep#1{\ifcat\relax\noexpand#1%
 \str=\expandafter{\the\str\noexpand#1}%
 \else\str=\expandafter{\the\str #1}\fi}
```

Note the following:

1. There is a (local) use of \obeyspaces. Without it, spaces are skipped when TEX determines the arguments of \scan.
2. Because no catcodes are changed, the four characters '#%{}' cannot appear in the argument of \VwriteM. A '#' in the argument will become '##' when the argument is absorbed. A '%' will send TEX looking for the rest of the argument on the next line. Unbalanced braces will cause an error message when the argument is absorbed. Balanced braces would be absorbed, would be used to nest groups in the argument, and will not appear on the file. For this reason, the use of \VwriteM is limited to cases where these characters do not appear in the strings to be written on file.
3. The \noexpand command adds an extra space. Thus '\VwriteM{\?M}' writes '\? M' on the file. To suppress the space, use \string instead of \noexpand in

   ```
   \str=\expandafter{\the\str\noexpand#1}%
   ```

   This may look better, but may give wrong results in some cases. A typical example is the expansion '\VwriteM{\bf M}', that would write '\bfM' on the file.
4. Our macros do not attempt to identify active characters. If the string includes any active characters, their expansions would be written on the file. It is, however, relatively easy to test for tokens of catcode 13 and insert a \noexpand in front of them.

Version 2: verbatim copying with \relax. A different way of avoiding expansion during file output is to temporarily turn an expandable control sequence into a non-expandable one. The simplest way of achieving this is to \let the control sequence be equal to \relax. Thus

```
\def\abc{...}
...
{\let\abc=\relax
 \immediate\write\aux{\abc}}
```

will write \abc on the file. Using this method we illustrate a different solution to the same problem. In this version, macro \onestep identifies all tokens in the string that are control sequences, and sets each equal to \relax.

After every control sequence in the string has been changed in this way, the string is written on a file. This version is similar to the previous one, the most important difference being that the final quantity being written on the file is '#1' and not the replacement text of a macro or the contents of a toks register. As a result, any braces in the argument will be written on the file (but see below for a subtle problem with braces).

```
\newwrite\out
\immediate\openout\out=filename
\def\VwriteN#1{{%
 \scan #1\end\immediate\write\out{#1}}}

\def\scan#1#2\end{\def\aux{#1}%
 \ifx\aux\empty
 \else
  \def\aux{#2}%
  \onestep{#1}%
  \ifx\aux\empty
  \else
   \scan#2\end
 \fi\fi}

\let\Let=\let
\def\onestep#1{\ifcat\relax\noexpand#1%
  \Let#1=\relax\fi}
```

The following points should be mentioned:

1. Macro `\VwriteN` has an extra pair of braces, so everything done in it is local. This way, the setting of control sequences to `\relax` is only temporary.
2. Imagine the string '`abc\let hjk\x`'. The control sequence `\let` is first identified, and is set to `\relax`. Later the control sequence `\x` is identified, but saying `\let\x=\relax` fails because `\let` is now equal to `\relax`. This is why the command '`\let\Let=\let`' has been added. Macro `\onestep` uses `\Let` instead of `\let`. Of course, a string such as '`...\Let...\x`' would cause the same problem, so this method cannot handle such strings.

   Exercise: What about the string '`...\Let`'?

   Answer: If `\Let` is the last control sequence (or the only one) in a string, our macros can handle it.
3. The above considerations apply to other commands used by `\onestep`, such as `\ifcat` and `\noexpand`. In principle, they should be redefined.
4. The '`#`' and active characters still cannot appear in the argument to `\VwriteN`, for the same reason as above.
5. Braces in the argument still must be balanced, but will be written on the file as mentioned earlier. There is another, subtle, problem associated with braces. Consider the expansion '`\VwriteN{{\bf M}}`'. At a certain step during the scanning, the argument of `\onestep` becomes the group '`\bf M`'. The `\noexpand#1` thus becomes '`\noexpand\bf M`' which typesets the '`M`'. The '`\Let#1=\relax`' becomes '`\Let\bf M=\relax`' which lets `\bf` to '`M`' and typesets the '`=`'. As a result, this version too, should only be used in limited cases.
6. Macro `\scan` does not use tail recursion because it has to expand either `\onestep` or itself with different parameters. As a result, each recursive expansion of `\scan` saves two `\fi`'s in the parameter stack, whose size is limited. A long argument will thus exceed TEX's capacity. This limitation is removed in the indexing example below.
7. This method works for an `\immediate\write` only. A non-immediate `\write` is executed in the OTR, where the various control sequences are no longer equal to `\relax`. This limitation, too, is removed in the indexing example below.

Indexing example. We again use indexing as an example to illustrate a general solution to the problem of verbatim copying. The macros for indexing discussed below are general and sophisticated but — in the opinion of the author — still readable. Among other things, they show how to handle general strings, and how to write the page number with the string. The basic task of the macros is to pick up certain items (flagged by a '`^`') and write them on the `.idx` file, which is later processed by `MakeIndex` (and, perhaps, other utilities) to create the final index.

The circumflex '`^`' is defined, as usual, to be the indexing character. It is declared active and is defined to be macro `\Caret`. A valid index item for the macros below must be one of the following:

- `^|abc|` where `abc` may contain any special characters (including unbalanced braces). The string `abc` is typeset verbatim, and also written verbatim on the `.idx` file.
- `^[abc]` where `abc` is as before. This is a 'silent' index item that's only written on the `.idx` file but is not typeset.
- `^{xyz}` where `xyz` may contain special characters (including a '`^`') but not a '`\`' (since it is sanitized during indexing), and not unbalanced braces. The string `xyz` will be typeset and written on the `.idx` file.

It is, however, invalid to say `^\abc` because the '`\`' is sanitized during indexing (one should say '`^[\abc]\abc`' instead). Also the argument of a macro cannot have index items, since all tokens in the argument get their catcodes when the argument is absorbed, and those catcodes cannot be changed later.

Here are examples of valid index items (see Refs. 1, 2 for the special meaning of the '`!`', the '`@`' and the parentheses).

```
^[character!special]   ^|\pop|
^[verbatim!listing|(]   ^[cmsy10]
^|^| ^[|^|!as an index]   ^[null@<null>]
^[|^^M|]   ^[|\TeX||)]   ^{#$%^&}
^{page break}   ^[\dvi\ file]
```

Exercise: How can one index a left (or right) brace?

Answer: It is invalid to write '`^{{}`'. An item of the form '`^[}]`' is fine, but it creates a record of the form '`\indexentry{}{1}`' on the `.idx` file, which cannot be properly read later. The item '`^|}|`' is fine but is not silent. A good choice is '`^[|}|]`'. It is silent and it writes '`\indexentry{|}|}{1}`' on the file. An even better choice is '`^[|}|@ (left brace)]`'. The part on the left of the '`@`' is the sort key, and the part on the right will be typeset (the print key).

If the '`^`' is used outside math mode, it becomes macro `\Caret`, which expands `\indexT`, which, in

turn, uses \futurelet to peek at the following token. If that token is a '|', macro \inxC is expanded. If it is a '[', macro \inxB is expanded; otherwise, \inxA is expanded. Each of these macros, in turn, expands \finidx which is responsible for the rest of the job.

Macro \finidx uses the primitive \meaning, so a short review of \meaning is necessary. The control sequence \meaning [213] creates a short explanation of its argument (such as 'the letter', 'macro' or 'math shift character'), followed by the tokens that make up the argument, with catcode 12 attached (except spaces, which get catcode 10). If the argument is a macro (e.g., '\def\abc#1;{A\B$#1^&\C_}'), then the commands '{\tt\meaning\abc}' result in 'macro:#1;->A\B $#1^&\C _'. The unnecessary tokens at the beginning can easily be stripped off by using > as a delimiter. This is done by macro '\def\strip#1>{}'.

Macro \finidx places the index item in a macro (\idxitem) and uses \meaning to obtain a string consisting of the individual tokens of the index item, each with catcodes as shown above. This string (together with other things) becomes the replacement text of macro \INDEX when \finidx says:

```
\edef\INDEX{\write\inx{%
 \string\indexentry%
 {\expandafter\strip\meaning\idxitem}%
 {\noexpand\folio}}}
```

A typical definition of \INDEX is thus:

```
\write\inx{\string\indexentry{abc}%
 {\noexpand\folio}}
```

\INDEX is then immediately expanded. Its expansion, however, is the \write command, which is not immediate, and is therefore saved, as a whatsit, in the MVL, to be executed in the OTR. Note that macro \INDEX is no longer needed, and can therefore be redefined when the next index entry is encountered.

Here is the complete set of indexing macros:

```
\def\Caret{\ifmmode\def\next{^}%
 \else\let\next=\indexT\fi\next}
\catcode`\^=\active \let^=\Caret
\def\indexT{\begingroup\sanitize%
 \makeother\\\obeyspaces%
 \makebgroup\{\makeegroup\}%
 \futurelet\new\inxcase}
\def\inxcase{\begingroup%
 \ifx\new|\aftergroup\tmpC
 \else\ifx\new[\aftergroup\tmpB
 \else\aftergroup\inxA \fi\fi \endgroup}
\def\tmpC{\makeother\}\makeother\{\inxC}
% deactivate braces during ^[...] & ^|...|
\def\tmpB{\makeother\}\makeother\{\inxB}
\def\inxA#1{\finidx{#1}#1}
\def\inxB[#1]{\finidx{#1}}
\def\inxC|#1|{\finidx{|#1|}|#1|}
\def\strip#1>{}
\long\def\finidx#1{\def\idxitem{#1}%
 \edef\INDEX{\write\inx{\string\indexentry%
  {\expandafter\strip\meaning\idxitem}%
  {\noexpand\folio}}}%
 \INDEX \endgroup}
```

A Warning. If a word is immediately followed by an index item, and the word happens to be the last one on the page, there is a chance that the item would be written on the index file with the number of the next page. The reason for this is that the indexing macros generate a (delayed) write, which becomes a whatsit in the MVL. Such a whatsit is executed in the OTR, when the page is shipped out. If the whatsit follows the last line of text on the page, there is a chance that the page builder would leave the whatsit in the MVL when preparing the current page. In such a case, the whatsit would become the first item on the next page.

A similar thing may happen if an index item immediately precedes the first word of a page. The item may end up being written on the index file with the number of the previous page.

A typical example is '...^[abc]xyz...'. If 'xyz' happens to be the first word on page 2, index item 'abc' may be written on the file with page number 1. If the document is short, the user may notice such a thing, and correct it by saying '...\hbox{^[abc]xyz}...'. This firmly attaches the index item to 'its' word (but then the word can no longer be hyphenated). If the word 'xyz' starts a paragraph, the user should change the mode to horizontal by '...\leavevmode\hbox{^[abc]xyz}...'

Verbatim listing

We now turn to the other aspect of verbatim namely, verbatim listing. The problem is to typeset verbatim any string of tokens, including spaces, braces, backslashes, or any other special characters. The problem is only important to people who write about TEX. Most other texts can get away with '\$' or '$\{$' to typeset any occasional special characters.

We start with a short review of interword spaces. A space (between words) is glue whose value is determined by the font designer. It is usually flexible but, in a fixed-space font, it should

be rigid (its value for font cmtt10, e.g., is 5.25pt). The size of a space is affected by the space factor, so that spaces following certain punctuation marks get more stretch (and sometimes even greater natural size). Naturally, this discussion applies to any character with catcode 10 (space being the only character assigned this catcode by INITEX [343]).

Consecutive spaces are treated as one space. To defeat this, the plain format offers macro \obeyspaces. The format starts by defining [351] '\def\space{␣}'. Thus \space is a macro whose replacement text is a normal space (affected by the space factor). Next, \obeyspaces is defined as a macro that declares the space active '\def\obeyspaces{\catcode`\␣=13}', and plain says (on [352])

{\obeyspaces\global\let␣=\space}

This means that when \obeyspaces is in effect (when the space is an active character) the space is defined as \space.

To get spaces that are not affected by the space factor, one of the following methods can be used:

- Change the sf codes of the punctuation marks to 1000 by means of \frenchspacing.
- Use a control space '\␣'. Control space [290] is a primitive that inserts glue equal to the interword space of the current font, regardless of the space factor. Defining the space as a control space is done by saying '{\obeyspaces\global\let␣=\␣}'.
- Assign nonzero values to \spaceskip and \xspaceskip.

Now we are ready for the verbatim macros. Four macros are discussed here, all extensions of macro \elp above. The aim is to develop macros that would typeset any given text, verbatim, in font cmtt10. The main problem is that the text may include special characters, such as '\' and '#', so these have to be turned off temporarily. Another problem is that the text has to be picked up line by line, and each line typeset individually. We shouldn't try to absorb the entire text as a macro argument since there may be too much of it. Other problems have to do with blank lines and consecutive spaces.

We start with macro \sanitize that's used, as usual, to change the catcodes of certain characters to 12 (other). It is similar to \dospecials [344].

```
\def\makeother#1{\catcode`#1=12\relax}
\def\sanitize{\makeother\%\makeother\#%
 \makeother\~\makeother\\\makeother\}%
 \makeother\{\makeother\&\makeother\$%
 \makeother\_\makeother\^\makeother\^^M%
 \makeother\ }
```

Now comes the main macro \ttverbatim. We tentatively start with the simple definition

```
\def\ttverbatim{\begingroup
 \sanitize\tt\gobble}
```

but the final definition below also contains '\def\par{\leavevmode\endgraf}' (in addition to a few other things). This is necessary because of blank lines. A blank line becomes a \par in the mouth, and \par has no effect in vertical mode. We thus have to switch to horizontal mode and do an \endgraf, which is the same as \par.

Macro \gobble gobbles up the end-of-line following the \ttverbatim, and expands \getline to get the first line of verbatim text. Without gobbling, \getline would read the end-of-line and translate it into an empty line in the verbatim listing.

Macro \getline gets one line of text (à la \elp), typesets it, executes a \par, and expands itself recursively. When it senses the end of the verbatim text, it should simply say '\endgroup' to revert to the original catcodes. The end of the text is a line containing just '\endverbatim' (without any preceding blanks), and the main problem facing \getline is to identify this line. The identification is done by means of \ifx, which compares two strings, stored in macros, character by character. The point is that an \ifx comparison is done by character code *and* category code. When the '\endverbatim' is read, sanitizing is in force, and the '\' has catcode 12 (the eleven letters have their normal catcode, 11).

We thus cannot simply define a macro '\def\endverb{\endverbatim}' and then compare '\ifx\endverb\aux', because the string in macro \endverb starts with '$\backslash_0$' instead of '$\backslash_{12}$'. The solution is to define \endverb in a group where the '\' has catcode 12. Thus

```
{\catcode`\*=0 \makeother\\%
 *gdef*endverb{\endverbatim}}
```

Now \getline can say \ifx\endverb\aux.

One of the verbatim methods below uses the vertical bar '|' to delimit small amounts of verbatim text. This is done by declaring the '|' active. Since we want to be able to include the '|' in verbatim listings, we sanitize it in \ttverbatim by saying '\makeother\|'.

After using the macros for several years, I was surprised one day to see a ? listed as ¿. A closer look revealed that it was the pair ?` that was listed as ¿. It took a while to figure out that, in the cmtt fonts, the combinations ?` and !` are considered

ligatures and are replaced by ¿ and ¡, respectively (Ref. 3, p. 36).

The solution is to declare the left quote active and to define it as macro \lq. This is why \ttverbatim and its relatives include the command '\catcode`\`=13', and why the code '{\catcode`\`=13\relax\gdef`{\relax\lq}}' is also necessary (see also [381]).

The definitions of the two macros should now be easy to understand.

```
\def\ttverbatim{\medskip\begingroup\tt%
 \def\par{\leavevmode\endgraf}%
 \catcode`\`=13\makeother\|\sanitize\gobble}
{\makeother\^^M\gdef\gobble^^M{\getline}%
 \gdef\getline#1^^M{\def\aux{#1}%
 \ifx\endverb\aux\let\next=\endgroup\medskip%
 \else#1\par\let\next=\getline\fi\next}%
 \obeyspaces\global\let␣=\␣\catcode`\`=13%
 \relax\gdef`{\relax\lq}}
{\catcode`\*=0 \makeother\\%
 *gdef*endverb{\endverbatim}}%
```

Exercise: The verbatim text above contains \endverbatim, but this string terminates verbatim listings. How was the text produced?

Answer: The string \endverbatim is only assigned its special meaning when it appears on a line by itself, with no preceding spaces, so in our case there was no problem. It is, however, possible to list an \endverbatim anywhere using the '|' (see below).

Readers trying these macros will very quickly discover that they typeset '␣' instead of spaces. This is because the space (whose character code is '40) has been sanitized (it is now a regular character, of catcode 12) and font cmtt10 has '␣' in position '40. This feature is sometimes desirable, but it is easy to modify \ttverbatim to get blank spaces in the verbatim listing.

The new macro is called \verbatim, and the main change is to say \obeyspaces instead of sanitizing the space. In verbatim listings, of course, we don't want the space to be affected by the space factor, so '{\obeyspaces\global\let␣=\␣}'.

```
\def\makeother#1{\catcode`#1=12\relax}
\def\sanitize{\makeother\%\makeother\#%
 \makeother\~\makeother\\\makeother\}%
 \makeother\{\makeother\&\makeother\$%
 \makeother\_\makeother\^\makeother\^^M}%
```

Macros \verbatim and \getline are defined by:

```
\def\verbatim{\medskip\begingroup\tt%
 \def\par{\leavevmode\endgraf}%
 \catcode`\`=13\sanitize\makeother\^^M%
 \makeother\|\obeyspaces\gobble}
{\makeother\^^M\gdef\gobble^^M{\getline}%
\gdef\getline#1^^M{\def\aux{#1}%
 \ifx\endverb\aux\let\next=\endgroup\medskip%
 \else#1\par\let\next=\getline\fi\next}%
 \obeyspaces\global\let␣=\␣%
 \catcode`\`=13\relax%
 \gdef`{\relax\lq}}
{\catcode`\*=0 \makeother\\%
 *gdef*endverb{\endverbatim}}%
```

Exercise: why do we have to place '{\obeyspaces\global\let␣=\␣}' outside the macros? It seems more elegant to have it included in the definition of \verbatim.

Answer: If we place it inside a macro, then the space following \let would get catcode 10 when the macro is defined. When the macro is expanded later, the \let command would fail, because it is followed by a catcode 10 token instead of by an active character.

Note the two \medskip commands. They create vertical spacing around the entire listing, and the first one also makes sure that the listing is done in vertical mode. They can be replaced, of course, by any vertical skip (flexible or rigid), depending on specific needs and personal taste.

Preventing line breaks. Each line of a verbatim listing is typeset by saying (in \getline) '#1\par'. The line becomes a paragraph and, if it is too wide, it may be broken. If this is not desirable, then the code above may be changed to \hbox{#1}. Macro \verbatim changes the mode to vertical, which means: (1) the boxes will be stacked vertically; (2) a wide box will not cause an 'overfull box' error.

Line numbers. The definition of \verbatim is now generalized to also typeset line numbers with the verbatim text. Macro \numverbatim below uses the same \sanitize as \verbatim, and a new count register is declared, to hold the current line number. The line numbers are typeset on the left margin, by means of an \llap, but this is easy to modify.

```
\newcount\verbline

\def\numverbatim{\medskip\begingroup\tt%
 \def\par{\leavevmode\endgraf}%
 \catcode`\`=13\sanitize\makeother\|%
 \obeyspaces\verbline=0
 \everypar{\advance\verbline1
 \llap{\sevenrm\the\verbline\␣\␣}}\gobble}
{\makeother\^^M\gdef\gobble^^M{\getline}%
\gdef\getline#1^^M{\def\aux{#1}%
 \ifx\endverb\aux\let\next=\endgroup\medskip%
```

```
 \else#1\par\let\next=\getline\fi\next}%
 \obeyspaces%
 \global\let␣=\␣\catcode`\`=13\relax%
 \gdef`{\relax\lq}}
{\catcode`\*=0 \makeother\\%
 *gdef*endverb{\endverbatim}}%
```

Verbatim in horizontal mode. The macros developed above are suitable for 'large' verbatim listings, involving several, or even many, lines of text. Such listings are normally done in vertical mode, between paragraphs. The next approach declares the vertical bar '|' active, and uses it to delimit small amounts of text (normally up to a line) that should be listed verbatim within a paragraph. This is convenient notation, commonly used, whose only disadvantage is that the '|' itself cannot appear in the text to be listed.

The first step is not to sanitize the space and the end-of-line:

```
\def\makeother#1{\catcode`#1=12\relax}
\def\sanitize{\makeother\%\makeother\#%
 \makeother\~\makeother\\\makeother\}%
 \makeother\{\makeother\&\makeother\$%
 \makeother\_\makeother\^}%
```

Next, the '|' is declared active, and is defined similar to `\verbatim` above. The main differences are:

- Macro `\moreverb` can pick up the entire text as its argument, since there is not much text.
- Instead of defining the space as a control space, we preempt the space by assigning non-zero values to `\spaceskip` and `\xspaceskip`.

```
\catcode`\|=13
\def|{\begingroup\obeyspaces%
 \catcode`\`=13\sanitize\moreverb}
\def\moreverb#1|{\tt\spaceskip=5.25pt%
 \xspaceskip=5.25pt\relax#1\endgroup}
```

(The value 5.25pt is the interword space of font cmtt10. If a different font is used, this value should be replaced by its interword space. An alternative is to use .51em, which gives good results in most sizes of cmtt.)

Exercise: Why is the `\relax` necessary after the `5.25pt`?

Answer: To terminate the glue specification. Without the `\relax`, if `#1` happens to be one of the words `plus` or `minus`, TEX would consider it to be part of the glue assigned to `\xspaceskip`, and would expect it to be followed by a number.

The reader should note that '|' cannot be used in the argument of a macro. If `\abc` is a macro, we cannot say, e.g., '`\abc{...|\xyz|...}`'. The reason is that all tokens in the argument get their catcodes assigned when the argument is absorbed, so '|' cannot change them later. Using '|' in a box, however, is okay.

Exercise: Change the definition of '|' to typeset '␣' instead of blank spaces.

Answer: Instead of using `\obeyspaces`, just sanitize the space (also the settings of `\spaceskip` and `\xspaceskip` are no longer necessary).

A different approach. Incidentally, there is a completely different approach to the problem of verbatim listing, using the primitive `\meaning`. The way this control sequence works has been reviewed earlier. To use it for verbatim listing, we simply say (compare with [382]):

```
\long\def\verbatim#1{{%
 \tt\expandafter\strip\meaning#1}}
\def\strip#1>{}
```

Since a macro is a token list, we can get verbatim listing of tokens this way, but with the following limitations: (1) extra spaces are automatically inserted by `\meaning` at certain points; (2) end of lines become spaces in the verbatim listing; (3) a single '`#`' cannot be included in the verbatim text (unless it is sanitized).

Fancy verbatim

Sometimes it is necessary to typeset parts of a verbatim listing in a different font, or to mix verbatim and non-verbatim text. Following are extensions of the verbatim macros, that can read and execute commands before starting on their main job. The commands are typically catcode changes but, in principle, can be anything. The commands are specified in two ways. Commands that should apply to all verbatim listings of a document are placed in the toks register `\everyverbatim`. Commands that should apply to just certain listings are placed between square brackets right following `\verbatim`, thus '`\verbatim[...]`'.

Macro `\verbatim` uses `\futurelet` to sneak a look at the token following the '`m`'. If this is a left bracket, the commands up to the right bracket are executed. Sanitization is done before the commands are executed, so the user can further modify the catcodes of sanitized characters. However, since the commands start with a '\', sanitization of this token should be deferred. The code below shows how `\verbatim` places the next token into `\nextc`, how `\options` expands `\readoptions` if this token is a '[', and how `\readoptions` scoops

up all the commands and executes them. Macro \preverbatim sanitizes the '\', and performs the other last minute tasks, before expanding \gobble.

```
\def\verbatim{\medskip\begingroup\sanitize%
 \the\everyverbatim%
 \makeother\^^M\futurelet\nextc\options}
\def\options{%
 \ifx[\nextc\let\next=\readoptions%
 \else\let\next=\preverbatim\fi\next}
\def\readoptions[#1]{#1 \preverbatim}
\def\preverbatim{\def\par{%
 \leavevmode\endgraf}\makeother\\%
 \makeother\^^M\tt\obeyspaces\gobble}
```

Note that the left quote is made active very late (together with the sanitization of the '\'). This means that the optional commands can use it in its original meaning, but they cannot change its catcode. It is possible to say, e.g., '\verbatim[\catcode`\*=11]', but something like '\verbatim[\makebgroup\`]' won't work because the left quote will be made active at a later point.

Advanced readers may easily change the macros such that the left quote would be made active early (perhaps by \sanitize). In such a case, the effect of

\verbatim[\catcode`*=11] can be achieved by defining
\def\makeletter#1{\catcode`#1=11 }, then saying
\verbatim[\makeletter*]

Similar remarks apply to the curly braces. Saying '\verbatim[\everypar={...}]' is wrong because the braces are sanitized early. The solution is to define '\def\temp{\everypar={...}}', then say '\verbatim[\temp]'.

The simplest example is '\verbatim[\parindent=0pt]' which prevents indentation in a specific listing. A more sophisticated example introduces the concept of meta code. The idea is that certain pieces of text in a verbatim listing may have to be typeset in a different font (we use cmr10). Such text is identified by enclosing it, e.g., in a pair of angle brackets '<>'. The following simple code implements this idea:

```
\def\enablemetacode{\makeactive\<}
{\enablemetacode \gdef<#1>{{\tenrm#1}}}
```

And the test:

```
\verbatim[\enablemetacode]
\halign{<...preamble...>\cr \beginCont
<...1st line...>\cr
<...>
<...last line...>\endCont\cr}
\endverbatim
```

produces

```
\halign{...preamble...\cr \beginCont
...1st line...\cr
...
...last line...\endCont\cr}
```

(It's easy to modify \enablemetacode to also typeset the brackets.) Next we introduce fancy comments. Suppose we want to typeset comments in a verbatim listing in italics. A comment is anything between a '%' and the end of line. Again, the following simple code is all that's needed to achieve this. It declares the '%' active (preempting the action of \sanitize), and defines it as a macro that typesets its argument in \it (the '%' itself can also be in \it, if desired).

```
\def\itcomments{\makeactive\%}
{\itcomments%
 \gdef%#1\par{\char"25{\it#1}\par}}
```

The test

```
\verbatim[\itcomments]
line 1 %comment 1
line 2  % Comment #2
line 3 % Note \this
\endverbatim
```

results in

```
line 1 %comment 1
line 2  % Comment #2
line 3 % Note "this
```

The next example typesets selected parts of a verbatim listing in \bf. The '!' is declared a temporary escape character, and the two parentheses, as temporary braces. The result of:

```
\verbatim[\makeescape\!
 \makebgroup\(\makeegroup\)]
  (!bf while) \lineno>\totalines
    \lineshiped:=\totalines
    (!bf extract) \temp (!bf from) \Bsav
    \totalines:=\totalines+\temp
  (!bf end while);
\endverbatim
```

is

```
  while \lineno>\totalines
    \lineshiped:=\totalines
     extract \temp  from \Bsav
    \totalines:=\totalines+\temp
  end while;
```

Note that the fancy commands between the square brackets should all fit on one line. They were broken

over two lines in the example above because of the narrow margin of *TUGboat*.

Sometimes, underlining is called for, to indicate keywords in a computer program. This can be achieved with:

```
\def\@#1@{\underbar{#1}}
\verbatim[\makeescape\!\catcode`\$=3]
!@var@ x, y, x1, x2: real;
x:=x1;
!@repeat@
y:=a*x+b;
point(round(x),round(y));
x:=x+0.01;
!@until@ x>x2;
\endverbatim
```

resulting in

```
var x, y, x1, x2: real;
x:=x1;
repeat
y:=a*x+b;
point(round(x),round(y));
x:=x+0.01;
until x>x2;
```

Sometimes a mixture of visible and blank spaces is required in the same verbatim listing. Here are two simple ways of doing this. The first one is:

```
{\tt\makeactive\!\gdef!{\char32 }}
\verbatim[\makeactive\!]
a 1!!2   3
x!4  5!!!6
\endverbatim
```

resulting in

```
a 1␣␣2   3
x␣4  5␣␣␣6
```

and the second one is:

```
{\makeactive\!\gdef!{\ }}
\verbatim[\vispacetrue\makeactive\!]
a 1!!2   3
x!4  5!!!6
\endverbatim
```

resulting in

```
a␣1  2␣␣␣3
x 4␣␣5   6
```

One more example, to convince the skeptics, that shows how math expressions can be placed inside a verbatim listing. We simply say:

```
\verbatim%
[\makeescape\!\catcode`\$=3 \catcode`\^=7]
prolog $!sum x^2$ epilog
\endverbatim
```

And the result is:

prolog $\sum\ x^2$ epilog

The concept of optional commands is powerful and can be extended to create verbatim listings that are numbered or that show visible spaces. This way, macros `\ttverbatim` and `\numverbatim` are no longer necessary and are replaced by `\verbatim[\vispacetrue]` and `\verbatim[\numbered]`, respectively.

The difference between macros `\verbatim` and `\ttverbatim` is that the former says '`\obeyspaces`', whereas the latter says '`\makeother\␣`'. We add a boolean variable `\ifvispace` that selects one of the choices above.

Macro `\numverbatim` says

```
\everypar{\advance\verbline1
 \llap{\sevenrm\the\verbline\ \ }}
```

We therefore define the two macros:

```
\def\numbered{\everypar{\advance\verbline1
 \llap{\sevenrm\the\verbline\ \ }}}
\def\notnumbered{\everypar{}}
```

that can turn the numbering on and off. The final version of `\verbatim` is shown below.

```
\def\verbatim{%
 \medskip\begingroup\tt\sanitize%
 \makeother\^^M\makeother\|%
 \futurelet\nextc\options}
\def\options{%
 \ifx[\nextc\let\next=\readoptions
 \else\let\next=\preverbatim\fi\next}
\def\readoptions[#1]{#1 \preverbatim}
\def\preverbatim{\def\par{%
 \leavevmode\endgraf}%
 \lastasks\ifvispace\makeother\ \else%
 \obeyspaces\fi\gobble}
```

The following tests are especially interesting:

```
\everyverbatim{\numbered\vispacetrue}
\verbatim
abc 123 \x %^?`!`
@#$% ^&  *(   )_
\endverbatim

\verbatim[\vispacefalse]
abc 123 \x %^?`!`
@#$% ^&  *(   )_
\endverbatim

\verbatim[\notnumbered]
abc 123 \x %^?`!`
@#$% ^&  *(   )_
```

```
\endverbatim
```

They result in:

```
1 abc␣123␣\x␣%^?`!`
2 @#$%␣^&␣␣*(␣␣␣)_

1 abc 123 \x %^?`!`
2 @#$% ^&  *(   )_
```

```
abc␣123␣\x␣%^?`!`
@#$%␣^&␣␣*(␣␣␣)_
```

The vertical bar can also take optional arguments. Below we show how to generalize the definition of '|', so things like

```
|[\enablemetacode]...<text>..|
|[\vispacetrue]..A B..|
```

will work.

The method is similar to the one used with `\verbatim`, with one difference: the backslash must be sanitized before `\futurelet` peeks at the next token. Consider the simple example '`|\abc|`'. The `\futurelet` will scoop up `\abc` as one (control sequence) token. Later, when `\moreverb` typesets its argument (when it says '`#1`'), there will be an error, since `\abc` is undefined.

If the `\futurelet` reads a '`[`', the backslash has to be restored (by '`\makeescape\\`'), so that macro `\readOptions` can read and execute the optional commands. Following that, macro `\preVerb` expands `\lastasks` to resanitize the backslash before the rest of the verbatim argument is read.

```
\makeactive\|
\def|{\begingroup\tt\obeyspaces%
 \sanitize\makeother\\%
 \futurelet\nextc\Voptions}
\def\Voptions{\ifx[\nextc\makeescape\\%
 \let\next=\readOptions
 \else\let\next=\preVerb\fi\next}
\def\readOptions[#1]{#1\relax\preVerb}
\def\preVerb{\lastasks\ifvispace%
 \makeother\ \else\obeyspaces\fi%
 \moreverb}
\def\moreverb#1|{\spaceskip=5.25pt%
 \xspaceskip=5.25pt\relax#1\endgroup}
```

Exercise: Why is the `\relax` necessary in macro `\readOptions` and why isn't it necessary in the similar macro `\readoptions` (expanded by `\verbatim`)?

Answer: `\readOptions` is expanded in horizontal mode, where spaces are sometimes significant, and `\readoptions`, in vertical mode, where spaces are ignored. The rule is that a space that's necessary as a separator is not typeset (does not become spurious). There is no strict need for a space after the '`#1`' since the `#` can be followed by one digit only (there can be at most nine parameters).

Exercise: Now that the '`[`' is special, how can we typeset it verbatim?

Answer: Easy, just turn it temporarily into a letter (catcode 11). The following `{\makeletter\[|[...|}` works nicely because the `\ifx[\nextc` compares `\nextc` to a left bracket with catcode 12.

Exercise: What is the effect, if any, of `|[\numbered]...|`?

Answer: No effect, since `\numbered` only changes `\everypar`, which is not used during vertical bar verbatim listing anyway. Since the change is done in a group, it is local.

Complete verbatim macros

Following is the complete code for all the verbatim macros necessary to implement the concepts discussed here. Note the new macro `\verbfile`. It can be used to list the contents of a given file verbatim. The argument '`#1`' is the name of the file. This is a simple modification of `\verbatim`, without optional commands (but see exercise below).

```
\newtoks\everyverbatim
\newcount\verbline
\newif\ifvispace \vispacefalse
\def\makeescape#1{\catcode`#1=0 }
\def\makebgroup#1{\catcode`#1=1 }
\def\makeegroup#1{\catcode`#1=2 }
% can have similar \make.. macros
% for catcodes 3--10
\def\makeletter#1{\catcode`#1=11 }
\def\makeother#1{\catcode`#1=12 }
\def\makeactive#1{\catcode`#1=13 }
\def\makecomment#1{\catcode`#1=14 }

\def\sanitize{\makeother\#\makeother\&%
 \makeother\%\makeother\~%
 \makeother\}\makeother\{\makeother\$%
 \makeother\_\makeother\^%
 \the\everyverbatim}%
\def\lastasks{\makeactive\`\makeother\\}

\def\verbatim{\medskip\begingroup\tt%
 \sanitize\makeother\^^M\makeother\|%
 \futurelet\nextc\options}
\def\options{%
 \ifx[\nextc\let\next=\readoptions
 \else\let\next=\preverbatim\fi\next}
\def\readoptions[#1]{#1 \preverbatim}
\def\preverbatim{%
 \def\par{\leavevmode\endgraf}%
```

```
 \lastasks\ifvispace\makeother\ \else%
 \obeyspaces\fi\gobble}

{\makeother\^^M%
 \gdef\gobble^^M{\getline}%
\gdef\getline#1^^M{\def\aux{#1}%
\ifx\endverb\aux
 \let\next=\endgroup\medskip%
\else#1\par\let\next=\getline\fi\next}%
\obeyspaces\global\let =\ \makeactive\`%
 \relax\gdef`{\relax\lq}}
{\makeescape\*\makeother\\%
 *gdef*endverb{\endverbatim}}%

\makeactive\|
\def|{\begingroup\tt\obeyspaces\sanitize%
 \makeother\\\futurelet\nextc\Voptions}
\def\Voptions{\ifx[\nextc\makeescape\\%
 \let\next=\readOptions
 \else\let\next=\preVerb\fi\next}
\def\readOptions[#1]{#1\relax\preVerb}
\def\preVerb{\lastasks\ifvispace%
 \makeother\ \else\obeyspaces\fi%
 \moreverb}
\def\moreverb#1|{\spaceskip=5.25pt
 \xspaceskip=5.25pt\relax#1\endgroup}

\def\verbfile#1 {\medskip\begingroup%
 \tt\def\par{\leavevmode\endgraf}%
 \sanitize\lastasks\makeother\|%
 \obeylines\obeyspaces\input#1\endgroup%
 \medskip}

\def\enablemetacode{\makeactive\<}
{\enablemetacode \gdef<#1>{{\tenrm#1}}}
\def\itcomments{\makeactive\%}
{\itcomments%
 \gdef%#1\par{\char"25{\it#1}\par}}
\def\numbered{\everypar{\advance\verbline1
 \llap{\sevenrm\the\verbline\ \ }}}
\def\notnumbered{\everypar{}}
```

Exercise: Why the `\relax` in
`'\makeactive\`\relax\gdef...'`
(normally a space is enough to terminate a number)?

Answer: Normally, a space following a number is considered a terminator, and is not printed. However, at this point, because of the `\obeyspaces`, the space is active (has catcode 13 instead of the normal 10), and is defined as a control space. It would therefore be typeset as a spurious space. This is especially annoying if the verbatim macros are part of a format file that is eventually dumped. We don't want such a file to create any typeset material.

Exercise: Extend the definition of
`\verbfile#1 {...}`
to detect and execute optional commands.

Answer:

```
\def\verbfile{%
 \medskip\begingroup\tt\sanitize%
 \makeother\|\futurelet\nextc\oPtions}
\def\oPtions{%
 \ifx[\nextc\let\next=\getoptions
 \else\let\next=\preverbfile\fi\next}
\def\getoptions[#1]{#1 \preverbfile}
\def\preverbfile#1 {%
 \def\par{\leavevmode\endgraf}%
 \lastasks\obeylines\ifvispace%
 \makeother\ \else\obeyspaces\fi%
 \input#1\endgroup\medskip}
```

A typical expansion is
`\verbfile[\numbered\vispacetrue]test`
where `test` is the name of the file (no space between the `]` and the file name).

References

1. Lamport, L., *MakeIndex: An Index Processor for LaTeX*, (available from archives carrying LaTeX stuff).
2. Chen, P., et al, *Index Preparation and Processing*, Software—Practice & Experience **18**(#9), 1988, pp. 897–915.
3. Knuth, D. E., *Computers and Typesetting*, Vol E, *Computer Modern Typefaces*, Addison-Wesley, 1986.

◇ David Salomon
California State University,
Northridge
Computer Science Department
Northridge, CA 91330-8281
dxs@secs.csun.edu