Update: Real Typesetting from Your Personal Computer

Alan Hoenig and Mitch Pfeffer

In Vol. 7, No. 3, we listed some sources of typesetting services for TEX users who need output from real typesetting machines. Some new information has turned up, and we are listing these new facts and figures according to the same rules applied in the original article: no favorites. And we ask the same consideration from readers — if anyone knows of other organizations that offer TEX typesetting services, please get in touch.

American Mathematical Society, P O Box 6248, Providence, RI 02940; (401) 272-9500
The AMS is now using an Autologic APS Micro-5 for its typesetting. At the present time the Society has AM, CM and Times Roman fonts available and within the next few months it expects to have many more typefaces from the Autologic library. Turnaround time varies depending on the size of the job but should be no more than a week for up to a 500 page job.

The AMS has also simplified its pricing structure. The charge for producing typeset output from your DVI file is $5 per page for the first 100 pages, $2.50 per page for all additional pages. The minimum charge for any job is $30. Files can be sent on VAX/VMS tapes or on IBM PC or Macintosh diskettes (although no PostScript extensions can be handled).

For scheduling purposes, the AMS asks that you contact them before submitting any jobs; talk to Regina Girouard.

Macros

Macros with Keyword Parameters

Wolfgang Appelt
Gesellschaft für Mathematik und
Datenverarbeitung, Sankt Augustin

TEX uses positional parameters for passing arguments to a macro. This has, as it is usually the case with positional parameters, two consequences: When calling a macro, which was defined to have parameters (`#1`, `#2`, ...),

1. the order of the arguments is important, i. e. it usually gives different results, if you write `\a{b}{c}` or `\a{c}{b}`, and
2. the number of the arguments must match the number of parameters in the definition of the macro.

There are, however, sometimes situations where the concept of keyword parameters is more adequate. Consider, for example, a `SetStyle` macro which may be used for modifying the formatting environment. The arguments, that can be passed to the macro, can be selected from a set of predefined keywords, say {`RAGGEDRIGHT`, `BOLD`, `ITALIC`, `NOINDENT`, ...}. The order of the arguments shall be unimportant and the number of the arguments shall be variable, i. e. the macro may be used as

```
\SetStyle(ITALIC)     or
\SetStyle(NOINDENT;ITALIC;RAGGEDRIGHT)
```

(If several arguments are present, they must be separated by a delimiter, for example by a ";".)

The following macros solve the problem (explanations afterwards):

```
\newif\if@more@keywords
\newif\if@keyword@matched

\def\SetStyle (#1){%
    \def\arguments{#1;\end}%
    \@more@keywordstrue
    \loop\expandafter\next@style@keyword
         \arguments
    \if@more@keywords\repeat}

\def\next@style@keyword #1;#2\end{%
    \def\next{#2}%
    \ifx\next\empty\@more@keywordsfalse
        \else\def\arguments{#2\end}\fi
    \@keyword@matchedfalse
    \compare@with@keyword #1<BOLD><\bf>%
    \if@keyword@matched
       \else\compare@with@keyword #1%
            <RAGGEDRIGHT><\raggedright>\fi
    \if@keyword@matched
       \else\compare@with@keyword #1%
            <ITALIC><\it>\fi
    \if@keyword@matched
       \else\compare@with@keyword #1%
            <NOINDENT><\parindent=0pt>\fi
    \if@keyword@matched\else
        \message{Unknown keyword #1!}\fi}

\def\compare@with@keyword #1<#2><#3>{%
    \def\next{#1}\def\keyword{#2}%
    \ifx\next\keyword
        \@keyword@matchedtrue #3\fi}
```

First we define two switches (`\if@more@keywords` and `\if@keyword@matched`) which are used within the macros. The `\SetStyle` macro has one parameter which is a keyword or a list of keywords, separated by a ";". The argument passed to the macro is saved in the macro `\argument` with a ";`\end`" added at the end. This will later on be used to detect the end of the argument. Then the macro starts calling `\next@style@keyword` within a loop as long as the switch `\@more@keywords` is `true`. The argument to `\next@style@keyword` is expanded before the macro is actually called, i. e. what the macro "sees" is something like

```
NOINDENT;ITALIC;RAGGEDRIGHT;\end
```

The macro `\next@style@keyword` splits the list of keywords into the first keyword, which is anything up to the first ";", and into the rest, which is anything up to the final `\end`. If the rest is empty, the switch `\@more@keywords` is set to `false`, so the loop in `\SetStyle` will stop. Otherwise the macro `\arguments` is redefined to contain the rest of the keywords with the "`\end`" added at the end again.

Then we start comparing the current keyword `#1` with the list of predefined keywords. This is done by calling the macro `\compare@with@keyword` several times, each time with a different specific keyword which is regarded a valid argument to `\SetStyle`. To avoid unnecessary calls of this macro when the current keyword was already found, we use the switch `\@keyword@matched`. If the current keyword was not recognized at all, when the list of the specific keywords is exhausted, an error message is written. The macro `\compare@with@keyword` is called with three arguments: the current keyword, a specific keyword and (usually) an action, that is to be performed, when the current keyword matches the specific one. The definition of `\compare@with@keyword` is simple: The first two parameters are compared. If they are identical, the switch `\@more@keywords` is set to `true` and the third parameter is "executed".

Expanding the set of valid keywords for the `\SetStyle` macro is trivial: It is only necessary to add furthers calls of `\compare@with@keyword`, each time with a new specific keyword, within the definition of `\next@style@keyword`. There is no restriction on the number of keywords, that can be used, i. e. the restriction, that a TEX macro must not have more than nine (positional) parameters, does not hold for keyword parameters.

In some applications a slightly different kind of keyword parameters is necessary. Consider, for example, a `\SetDimensions` macro, which shall be used for modifying some parameters controlling the size and positioning of a page. The macro may, for example, be used as

```
\SetDimensions(VSIZE=1.5\char92hsize)   or
\SetDimensions(HSIZE=20pc;HOFFSET=10pt)
```

Now each keyword has an associated value, which shall be passed to some "attribute", denoted by the keyword. This case can be handled by the following macros:

```
\def\SetDimensions (#1){%
    \def\arguments{#1;\end}%
    \@more@keywordstrue
    \loop\expandafter\next@setdim@keyword
         \arguments
    \if@more@keywords\repeat}

\def\next@setdim@keyword #1;#2\end{%
    \def\next{#2}%
    \ifx\next\empty\@more@keywordsfalse
        \else\def\arguments{#2\end}\fi
    \@keyword@matchedfalse
    \compare@with@attribute #1%
            <HSIZE><\hsize=\value>%
    \if@keyword@matched
       \else\compare@with@attribute #1%
            <VSIZE><\vsize=\value>\fi
    \if@keyword@matched
       \else\compare@with@attribute #1%
            <HOFFSET><\hoffset=\value>\fi
    \if@keyword@matched
       \else\compare@with@attribute #1%
            <VOFFSET><\voffset=\value>\fi
    \if@keyword@matched\else
        \message{Unknown keyword #1!}\fi}

\def\compare@with@attribute#1=#2<#3><#4>{%
    \def\next{#1}\def\keyword{#3}%
    \ifx\next\keyword\@keyword@matchedtrue
        \def\value{#2}#4\fi}
```

The macros are rather similar to the previous ones. The main difference to the example above is the macro `\compare@with@attribute` which is used instead of the former `\compare@with@keyword`. The macro is called with three arguments as in the previous example, but it has four parameters in its definition, the first two being separated by a "=". This serves for splitting the first argument, which might, for example, be

```
VSIZE=1.5\hsize
```

into the "attribute name" and the "attribute value". If the attribute name matches the third parameter (or — looking at the call of the macro — the second argument), the switch `\@keyword@matched` is set to

true and the meaning of \value is defined as the attribute value.

Notice the macro \value: When it is passed as an argument to \compare@with@attribute it is still undefined. In other words, we have the funny case of a macro which — to some extent — defines the arguments, that it receives, itself.

The two examples above show rather simple applications of keyword parameters without great practical value. They should primarily be regarded as an explanation of the basic ideas how such macros can be written. In practice further extensions may be necessary. One extension may be the mixture of positional and keyword parameters, another one the definition of macros, where the keywords in the argument list may have to be reordered before they get interpreted.

The discussion on positional versus keyword parameters has a long tradition in computer science and common understanding is probably, that keyword parameters are preferable to positional ones in many cases. Also several document processing systems, e. g. Reid's *SCRIBE* system (B. K. Reid: *SCRIBE — Introductory User's Manual*, Unilogic Ltd., Pittsburgh, 1980), make use of keyword parameters to some extent. (There are even a few features in LaTeX which look like keyword parameters though Lamport does not use this terminology. See, for example, the *options* that can be given with a \documentstyle command.)

Using the concept of keywords parameters can probably lead to macro packages with user interfaces, which look quite different from existing ones and might be preferred by many users. Maybe even the writing of "bridgeware" macro packages to other formatting languages, for example a macro package that makes (at least certain classes of) *SCRIBE* documents processable by TeX, might become easier.

When I first thought about keyword parameters I was surprised, that it took only a few hours to write down some macros that solved the problem. So, if after all the examples above may show nothing, they at least prove once again the flexibility and power of TeX's macro language.

\expandafter vs. \let and \def in Conditionals and a Generalization of PLAIN's \loop

Alois Kabelschacht
Max-Planck-Institut für Physik

Conditionals with \expandafter

Sometimes the replacement text for a TeX macro should end with one or another macro call, depending on a condition. The trivial solution

```
... \if... ... \aa \else ... \bb \fi
```

works only if neither \aa nor \bb needs an argument. Otherwise a more complicated construction such as the following example from plain.tex is needed:

```
\def\ph@nt{\ifmmode
    \def\next{\mathpalette\mathph@nt}%
  \else\let\next\makeph@nt\fi\next}
```

There is the alternative:

```
\def\ph@nt{\ifmmode
    \expandafter\mathpalette
    \expandafter\mathph@nt
  \else\expandafter\makeph@nt\fi}
```

which uses the fact that the expansion of both \else ... \fi and \fi is empty. This alternative is definitely shorter (by 4 tokens) and as far as I can see not slower. It has the further advantage that it also works if expandable tokens are expanded but no commands are digested (e.g. in the replacement text for \edef). The alternative construction is clearly even more economical in such cases where one of the branches would otherwise contain a '\let\next\relax'.

A generalization of PLAIN's \loop macro

Using the above idea one could e.g. replace PLAIN's definition of \iterate (used in conjunction with \loop):

```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body \let\next\iterate
    \else\let\next\relax\fi \next}
\let\repeat=\fi % this makes
      % \loop...\if...\repeat skippable
```

by

```
\def\iterate{\body
    \expandafter\iterate\else\fi}
```

Finally, omitting the \else and rearranging things a bit one obtains

```
\def\loop#1\repeat{\def\iterate
      {#1\expandafter\iterate\fi}%
  \iterate \let\iterate\relax}
```