TEX beauties and oddities

Abstract
The BachoTEX 2009 conference continued the Pearls of TEX Programming open session introduced in 2005 during which volunteers present TEX-related tricks and shorties.

A permanent call for TEX pearls

What is wanted:

- short TEX or MetaPost macro/macros (half A4 page or half a screen at most),
- the code should be generic; potentially understandable by plain-oriented users,
- results need not be useful or serious, but language-specific, tricky, preferably non-obvious,
- obscure oddities, weird TEX behaviour, dirty and risky tricks and traps are also welcome,
- the code should be explainable in a couple of minutes.

The already collected pearls can be found at http://www.gust.org.pl/pearls. All pearl-divers and pearlgrowers are kindly asked to send the pearl-candidates to pearls@gust.org.pl, where Paweł Jackowski, our pearl-collector, is waiting impatiently. The pearls market-place is active during the entire year, not just before the annual BachoTEX Conference.

Note: The person submitting pearl proposals and/or participating in the BachoTEX pearls session does not need to be the inventor. Well known hits are also welcome, unless already presented at one of our sessions.

Since some seasoned TEX programmers felt indignant of calling ugly TEX constructs 'Pearls of TEX programming', we decided not to irritate them any longer. We hope they will accept 'TEX beauties and oddities' as the session title.

If you yourself have something that fits the bill, please consider. If you know somebody's work that does, please let us know, we will contact the person. We await your contributions even if you are unable to attend the conference. In such a case you are free either to elect one of the participants to present your work or 'leave the proof to the gentle reader' (cf. e.g. http://www.aurora.edu/mathematics/bhaskara .htm).

Needless to say that all contributions will be published in a separate section of the conference proceedings, possibly also reprinted in different TEX bulletins.

Scary space

Hans Hagen & Taco Hoekwater

In pure TeX

```
    \show\
    \show\ %
```

gives different logs on Hans' and Taco's machines (Hans' is on the left)

```
    **space.tex                            **space.tex
    (./space.tex                           (./space.tex
    > \                                    > \^^M=macro:
    =macro:                                ->\ .
    ->\ .                                  l.1 \show\
    l.1 \show\
                                           ?
    ?                                      > \ =\ .
    > \ =\ .                               l.2 \show \
    l.2 \show \                                        %
                %                          ?
    ?                                       )
     )                                     No pages of output.
    No pages of output.
```

The visualization of a `^^M` depends of the platform but since there's definitely a newline involved we need to take care of it.

When parsing the input the following happens (this is mentioned in one of the dangerous bends in the TEXbook):

```
     \let\x\ <newline> => \let\x\<endlinechar>
```

This means that when you want to store the meaning of this primitive, you need to make sure that TEX explicitly sees a space instead of a newline. So we get:

```
    \let\normalspaceprimitive=\ % space-comment is really needed
```

In ConTeXt this is used for:

```
    \unexpanded\def
       \ {\mathortext\normalspaceprimitive{\dontleavehmode\space}}
```

If you don't use the explicit space a simple

```
    $\ $
```

will execute `\^^M`. In Plain TeX (and in ConTeXt) we have:

```
    \def\^^M{\ } % control <return> = control <space>
```

So this will result in a loop.

Null control sequence

Hans Hagen & Taco Hoekwater

When you do:

```
\endlinechar=-1
\let\x\
```

the macro `\x` is undefined...

Actually `\x` becomes equal to the 'null control sequence' that you would get from

```
\expandafter\def\csname\endcsname{}
```

but that is usually undefined.

And you can even use this effect to assign to the null control sequence without needing `\expandafter`:

```
\endlinechar=-1
\gdef\
    {\message{NULL CS}}

\csname\endcsname
```

$$: empty formula or unmatched display

Hans Hagen & Taco Hoekwater

When you try the following under TEX'S normal catcode regime, you will get an error:

```
$$ $21-09$
```

The message is:

```
! Display math should end with $$.
 <to be read again>
                  2
 l.7 $$ $2
          1-09$
```

But how about this then:

```
\halign{#&#\cr $$ & $21-09$\cr}
```

It magically works! The actual effect is similar to

```
\hbox{$$} % or
\hbox{${}$}
```

In words of the TEXbook (chapter 25, page 287):

“One consequence of these rules is that `$$' in restricted horizontal mode simply yields an empty math formula.”

<macro> macro

Philip Taylor

Typesetting a multi-lingual document, even something as simple as a Christmas letter, can be time-consuming and error-prone if the embedded languages make frequent use of diacritics. To eliminate both of these problems, I wrote a macro called `\macro` which enables me to encapsulate all of the tricky words and phrases into macros whose names are (normally) identical to the words or phrases but without the corresponding diacritics.

The following code implements the `\macro` macro, and is followed by some sample definitions and applied occurrences.

```
\catcode`\<=\active
\def<#1>{%
     % cf. Bernd Raichle: check if defined, no side effects (2006)
     \if \csname macro:#1\endcsname \relax
            {\bf {$\ll$}#1{$\gg$}}%
     \else
            \csname macro:#1\endcsname
     \fi}

\def\macro#1#2{\expandafter\def\csname macro:#1\endcsname{#2}}

\macro {Zhou Shang Zhi}{Zh\=ou Sh\`ang Zh\=\i}
\macro {Shangzhi}{Sh\`angzh\=\i}
\macro {Sai Gon}{S\`ai G\`on}
\macro {HCM}{H\raise 0.5ex \rlap {\` }\^o Ch\'\i{} Minh}
\macro {Mui Ne}{M\~ui N\'e}
%\macro {Le}{L\rlap {\d e}\^e}

On a~happier note, the year started with both Khanh \&~I~being
invited to spend time with one of my former Chinese teachers,
<Zhou Shang Zhi>, and his family in Kyoto, Japan. <Shangzhi>
was there for one year, teaching at a~local university, and
the last three months were effectively a~holiday for him with
very few formal duties. Knowing that we might like to visit
Kyoto, <Shangzhi> very kindly invited both of us, which we
accepted with great pleasure.

Khanh's journey commenced with a~flight to <Sai Gon>
(``<HCM> City''), from where she took a~'bus south to <Mui Ne>
(a~distance of some 100 miles or so), where her sister
<Le>~Hoa had booked her into a~very posh hotel by the
beach. Once in <Mui Ne>, Khanh hired a~moped driver.
```

On a happier note, the year started with both Khanh & I being invited to spend time with one of my former Chinese teachers, Zhōu Shàng Zhī, and his family in Kyoto, Japan. Shàngzhī was there for one year, teaching at a local university, and the last three months were effectively a holiday for him with very few formal duties. Knowing that we might like to visit Kyoto, Shàngzhī very kindly invited both of us, which we accepted with great pleasure.

Khanh's journey commenced with a flight to Sài Gòn ("Hồ Chí Minh City"), from where she took a 'bus south to Mũi Né (a distance of some 100 miles or so), where her sister ≪**Le**≫ Hoa had booked her into a very posh hotel by the beach. Once in Mũi Né, Khanh hired a moped driver.

UTF-8 support detection

Arthur Reutenauer

When you need to detect if you are running an extension of TEX that supports UTF-8 input, you can use an extensive approach by making the list of engines that could be concerned, and check for particular control sequences like `\XeTeXversion` for XETEX, or `\directlua` for luaTEX. But you can also simply check for UTF-8 directly, by counting the bytes:

Take Τ, the letter Tau from the Greek alphabet, not the Latin one that looks like it. In UTF-8, its encoding form uses two bytes, which means it is read as two characters by 8-bit TEX engines, but only one by UTF-8 engines. Hence, the following lines detect UTF-8 engines:

```
\def\testengine#1#2!{\def\secondarg{#2}}
```

That's Tau (as in TEX),

```
\testengine Τ!\relax

UTF-8
\ifx\secondarg\empty
       is % We're UTF-8
\else
       not % We're 8-bit
\fi
supported.
```

Abba Don

Grzegorz Murzynowski

What is and what is not a number for TEX? Adoremus magna et mirabilia opera pappæ Knuth!

```
\ifnum 666>'0888${}-222 = 2\times32\times37={}$DCLXVI
   (all the Roman digits except the largest)\fi

\ifnum 666>"000ecce Angelus Pulcherissimus regnavit!\fi

\ifnum 666>"0000ABBA Father call I from the deepest of my s***
\else Breke kekk, breke kekk!\fi

\ifnum 11254493="ABBADDON($\aleph_0$)\fi
```

Note that `A`, `B`, `c`, `D` and `e` are (in some contexts) hexadecimal digits and (in those contexts) `0xecce` = 60622 and (in some other contexts) Abbaddon is the name of the Angel of Extinction.

inlinedef: a general recursive token scanner with callbacks

Stephen Hicks

There have been several discussions about uses of `\expandafter` that border on the ridiculous, with as many as fifteen in a row found in actual TeX input files! Additionally, trying to expand past macro parameters #1 causes problems because there is no guarantee that #1 is a single token. It would instead be nice to insert something right before a single token we want to expand far in advance. A slightly more general problem is to scan tokens in the input stream while preserving spaces and grouping.

```
\let\xa\expandafter

\def\scan{\futurelet\foo\switch}
\def\switch{%
  \let\next\normal
  \ifcat\noexpand\foo\space \let\next\dospace\fi
  \ifcat\noexpand\foo\bgroup \let\next\trygroup\fi
  \ifcat\noexpand\foo\relax \try{&\meaning\foo}\fi
  \next}
\def\try#1{\ifcsname #1\endcsname\xa
           \let\xa\next \csname #1\endcsname\fi}
\def\dospace{\toks0\xa{\the\toks\xa0 \space}\xa\scan\unspace}

\xa\def\xa\unspace\space{}
\long\def\trygroup#1#{%
  \def\temp{#1}\xa\let\xa\next
  \ifx\temp\empty\recurse\else \normal\fi\next#1}
\long\def\recurse#1{%
  \begingroup\toks0{}\scan#1\END{}\xa\endgroup\xa
  \toks\xa0\xa\xa\xa{\xa\the\xa\toks\xa0\xa{\the\toks0}}\scan}
\long\def\normal#1{\toks0\xa{\the\toks0 #1}\scan}

\def\callback#1#2#{%
   \def#1{\noexpand#1}\xa\def\csname&\meaning#1\endcsname#2}
```

We can set up a few callbacks, e.g. `\END` to end scanning, and `\EXPAND` to expand the next token:

```
\callback\END#1{}
\callback\EXPAND#1{\expandafter\scan}
```

And now we can get arbitrary tokens from the input stream into `\toks0` using

```
\def\baz{!}
\scan foo {bar \EXPAND\baz} \baz \END
\message{\the\toks0} % foo\space {bar\space !}\space \baz
```

This can be made more general in several ways: if we don't check `\ifcat \noexpand\foo\relax` then we can execute callbacks on arbitrary tokens, including spaces and grouping symbols. Of course this slows things down quite a bit further, which brings me to the main disadvantage of this approach: it takes about 25 times as long as a simple string of `\expandafter`'s, and is therefore not suitable for inner loops. But the code it allows us to write, as long as efficiency isn't important, is much more readable.