Enjoy TeX pearls diving!

TeX Pearls Session – BachoTeX 2006

The BachoTeX 2006 conference continues the Pearls of TeX Programming open session introduced in 2005 during which volunteers present TeX-related tricks and shorties.

What is wanted:

– short TeX, METAFONT 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.

Already collected pearls can be found at `http://www.gust.org.pl/pearls`. All pearl-divers and pearl-growers 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 pearls session needs not necessarily be the inventor. Well known hits are also welcome, unless already presented at one of our sessions.

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`).

Frank Mittelbach

Is there a font?

The following code allows to check, if some font (`tfm` metric) is available.

```
\batchmode
\font\X=<font name>%
\errorstopmode
 \ifx\X\nullfont
    <font not available>
 \else
    <use the font>
 \fi
```

Having ε-TeX we can switch between modes using handy `\interactionmode` primitive.

Paweł Jackowski

\relax ex machina

Guess what is the meaning of macros in the following cases:

```
\edef\stra{\csname undefined\endcsname}
\edef\strb{\ifnum0=1\else\fi}
\edef\strc{\ifnum0=0\else\fi}
\edef\strd{\relax}

\meaning\stra % -> \undefined (plus side effect alike \let\undefined\relax)
\meaning\strb % -> (empty)
\meaning\strc % -> \relax
\meaning\strd % -> \relax
```

. . . and can you explain why `\ifx\strc\strd` is false, although both control sequences have the same meaning?

```
\ifx\strc\strd true\else false\fi \message{?\strc?\strd?}
```

Since the behavior of is somewhat weird, I've learned to dislike the acting as the universal string delimiter in cases such as

```
\def\gobbler#1\relax{}
\expandafter\gobbler\strc whatever \relax

\def\iterator#1{%
  \ifx\relax#1\else\message{Can you see that #1?!}%
  \expandafter\iterator\fi}
\expandafter \iterator \stra whatever \relax
```

Instead I propose

```
\def\endstr{\noexpand\endstr}
```

– no endless loop in spite of recursion (expands to itself)
– alike **\relax**, **\endstr** returns nothing (a sort of...) if typeset
– side effect is that **\endstr** stops assignments (as **\relax** do)
– `\ifx\endstr\relax` is false, so we can distinguish them
– but be careful: `\if\endstr\relax` is true

Bernd Raichle

\relax ex machina II

Question: Is the following TeX input correct, i.e, will TeX abort with an error message or not?

```
\hbox \relax \relax {haha}
\hbox to \hsize \relax \relax {hehe}
\moveright .25\hsize \relax \relax \hbox{hihi}

\toks0 = \relax \relax {hoho}
\toksdef\tlist = 2  \tlist = \relax \relax {huhu}

\setbox0 = \relax \hbox \relax {haha}

\global \relax \global \relax \long \relax \def\foo{oh!}

$$ \halign{\kern 20pt #\cr
          \noalign \relax \relax {aha}\cr
          oho\cr} \relax $$

$ \left \relax ( 1 \mathord \relax {+} 2 ^ \relax {3} \right \relax ) $

a\leaders \hbox \relax {\TeX} \relax \hskip .5\hsize b

bla \vadjust \relax \relax {foo} bla.

B\accent 127\relax \relax \relax \relax a%
\discretionary \relax {k-} \relax {k} \relax {ck}%
er.
```

Answer: Every token above is not needed. Nonetheless these tokens do not cause any error, they are allowed and ignored at the shown places.

If you browse through the TeX source code file `tex.web` you will propably have seen the web chunk named `@<Get the next non-blank non-relax non-call token@>`. This web chunk is used in the procedure `scan_left_brace`, in `box_end` for leaders, in `scan_box`, in `scan_math`, in `scan_delimiter`, in `prefixed_command`, in assignments of `\toks` or `\toksdef` tokens after the equal sign, in `do_assignments`. These procedures are used to scan the input for some TeX primitive constructs. The effect is that spaces and `\relax` tokens can be inserted at various places without harm because they are ignored.

Bernd Raichle

Read only first line of an input file

You have a TeX input file with more than one line:

```
test file 1 - line 1 \count0=42
test file 1 - line 2 \count0=31415
test file 1 - line 3 \count0=1
... more lines ...
```

You can use the following trick, if you want to process only the first line of this file:

```
\endinput \input testfile
```

This will produce the following output:

```
test file 1 - line 1
```

Background: The primitive `\endinput` does not close the current file immediately. It sets a TeX internal flag that indicates that if the next end-of-line is processed the current file is closed. Thus all the tokens after the `\endinput` until the end of the line are processed. If you start reading a new file after `\endinput` this file will become the current file and it will be closed after the first end of line is seen.

```
\immediate\openout0=\jobname.out
\immediate\write0{test file 1 - line 1 \count0=42}
\immediate\write0{test file 1 - line 2 \count0=31415}
\immediate\write0{test file 1 - line 3 \count0=1}
\immediate\closeout0
...
\endinput\input zz.out (\the\count0) \end
```

As a result we get:

... test file 1 – line 1 (42)

Bernd Raichle

\input file inside a macro definition

If you have a file and want to read its contents in a macro definition, TeX does not allow to simply use

```
\edef\foo{\input file }
```

because at the end of a file TeX checks if you are inside an incomplete \if statement or \def macro definition and outputs the error message

```
! File ended while scanning text/definition of ...
```

Nonetheless there is a non-trivial way to avoid this error message:

```
\immediate\openout15=bt-pp-r4.tex
\immediate\write15{Another TeX}
\immediate\write15{Pearl\string\noexpand}
\immediate\closeout15

\edef\foo{Yet \input bt-pp-r4.tex !}
\message{\meaning\foo}
```

The primitive \noexpand does all the magic and you will get

```
macro:->Yet Another TeX Pearl!
```

without any error message.

Question: Why has \noexpand this effect?

If you are using ε-TeX, you can set its new special token list \everyeof to \noexpand to achieve this effect without changing the file.

Taco Hoekwater

4 endless lines

The end of a line in an `\input`-ed file normally creates a space in the output, TeX appends a character with the current value of `\endlinechar` to each that character is later converted to a space.

```
\immediate\write18
   {echo -n Y >bla.tex}
X\input bla Z
% gives XY Z
```

If you do not want that space, for instance because you want to typeset the of the file in-line, then there are a number of options. Two are usable writing, and two others during type reading.

1. Writing a percent sign to the end of the line works

```
\immediate\write18
   {echo -n Y\letterpercent >bla.tex}
X\input bla Z
% gives XYZ
```

2. Another option is ending the written line with `\relax` or a similar space-gobbling command.

```
\immediate\write18
   {echo -n Y\letterbackslash\letterbackslash relax >bla.tex}
X\input bla Z
% gives XY\relax Z
```

3. Temporarily setting `\endlinechar` to an impossible value like -1 is a possibility

```
\immediate\write18
   {echo -n Y >bla.tex}
X{\endlinechar=-1 \input bla }Z
% gives XYZ
```

4. Changing the catcode of the current `\endlinechar` to 9 (ignored) also works

```
\immediate\write18
   {echo -n Y >bla.tex}
X{\catcode'\^^M=9 \input bla }Z
% gives XYZ
```

Bernd Raichle

Check if defined, no side effects

LaTeX and other macro packages include a test if a control sequence is already defined using `\csname...\endcsname`. In principle the following definition is used:

```
\def\@ifundefined#1#2#3{%
  \expandafter\ifx\csname #1\endcsname\relax#2\else#3\fi}
```

The use of `\csname...\endcsname` has the side effect that an undefined control sequence will be defined as `\relax` and a simple test using `\ifx<control sequence>\undefined` will fail.
To avoid this side effect, you can change the definition above introducing a group "around" `\csname`:

```
\def\@ifundefined#1#2#3{%
  \begingroup \expandafter\expandafter\expandafter \endgroup
  \expandafter\ifx\csname #1\endcsname\relax#2\else#3\fi}
```

Note: This definition is not fully expandable, thus it will fail in expansion-only contexts where the original definition will work.
If you are using ε-TeX, the new `\ifcsname...\endcsname` primitive can be used instead.

During the presentation of this pearl at the BachoTeX 2006 pearls session, two alternatives to this trick has been proposed. Worthy to note, that the macro above use more `\expandafter`'s then actually needed. Once we say

```
\def\@ifundefined#1#2#3{%
  \begingroup \expandafter
  \ifx\csname#1\endcsname\relax\endgroup#2\else\endgroup#3\fi}
```

we get exactly the same effect in a bit more efficient way. But there is one more sticky problem in all the constructions above; every control sequence defined as `\relax` (and `\relax` itself) is treated as undefined. The solution seems to be

```
\def\@ifundefined#1#2#3{%
  \begingroup \expandafter \endgroup \expandafter
  \ifx\csname#1\endcsname\undefined#2\else#3\fi}
```

Here we expand `\csname...\endcsname` before `\endgroup`, but the condition is fixed outside the group. Thus, `\undefined` instead of `\relax` can be used for comparison.

Bernd Raichle
Global assignments done locally

Sometimes it is necessary to define a macro under a changed input regime, e.g, using different category codes or another end line character. Usually this is done inside a group to keep the changes locally and the macro or the token is defined `\global`'ly.

In `plain.tex` you can find the following examples:

```
\catcode`\@=11
```

1. Helper macro for `\newif`:

```
{\uccode`1=`i \uccode`2=`f \uppercase{\gdef\if@12{}}} % `if' is required
```

2. Definitions of `\obeylines` and `\obeyspaces`:

```
{\catcode`\^^M=\active % these lines must end with %
  \gdef\obeylines{\catcode`\^^M\active \let^^M\par}%
  \global\let^^M\par} % this is in case ^^M appears in a \write
{\obeyspaces\global\let =\space}
```

3. Definition of `\getf@ctor`:

```
{\catcode`p=12 \catcode`t=12 \gdef\\#1pt{#1}} \let\getf@ctor=\\
```

4. Math definitions for primes and underscore:

```
{\catcode`\'=\active \gdef'{^\bgroup\prim@s}}
{\catcode`\_=\active \global\let_=\_} % _ in math is either subscript or \_
```

By placing the begin and end of group tokens in a bit "unusual" way using a temporary token register assignment or a macro definition, all these assignments can be done locally:

1. Helper macro for `\newif`:

```
\begingroup \uccode`1=`i \uccode`2=`f
\uppercase{\endgroup\def\if@12{}}% `i'+`f' as delimited arguments are required
```

2. Definitions of `\obeylines` and `\obeyspaces`:

```
\begingroup \endlinechar=-1 \catcode`\^^M=\active
  \toks0={\endgroup
    \def\obeylines{\catcode`\^^M\active \let^^M\par}%
    \let^^M=\par  % this is in case ^^M appears in a \write
  }\the\toks0\relax
\begingroup \obeyspaces\def\x{\endgroup\let =\space}\x
```

3. Definition of `\getf@ctor`:

```
\begingroup \catcode`P=12 \catcode`T=12
  % \lccode`P=`p \lccode`T=`t % is default setting
  \lowercase{\endgroup\def\getf@ctor#1PT{#1}}%  `p'+`t' with catcode `other'
```

4. Math definitions for primes and underscore:

```
\begingroup \catcode`\'=\active
  \def\x{\endgroup\def'{^\bgroup\prim@s}}\x
\begingroup \catcode`\_=\active
  \def\x{\endgroup \let_=\_}\x% _ in math is either subscript or \_
```

There is no advantage of a local definition for the shown plain.tex cases but if you want to do similar definitions within a group, the shown technique can be very helpful.

Note: If you want to define macros with arguments it is better to use a token register assignment because you have to double the hash mark as macro parameter character inside the macro definition text.

Hans Hagen

\lccode builds a hash table

Wybo Dekker (NTG) asked on the TEX-NL list how to construct a macro where in the name a '1' would be replaced by an 'A'. This involves a conversion and TeX does not ship with that many straightforward conversion features. His approach involved the \char primitive but failed for the (maybe not that obvious) reason that this primitive results in a character node and in expansions (as in an \edef) remains as it is: a reference to a specific character (slot) in the font used at the moment when it is applied.

```
\Name{test}{1}{Hello}
\Name{test}{2}{World!}
\testA\ \testB % this should give: Hello World!
```

A rather convenient way to remap characters is using the \lccode or \uccode primitives in combination with \lowercase or \uppercase. The following solution is a bit optimized in the sense that we expand the temporary variable before we end the group. An alternative is to define \temp globally and to omit the first \expandafter. Watch how we apply \lowercase to the whole definition; using the \lowercase inside an \edef would not work. Instead of using a loop an dsome calculations, we directly assign the codes. The 0/J case is an exception anyway.

```
\long\def\Name#1#2#3%
 {\bgroup
  \lccode'1='A \lccode'6='F
  \lccode'2='B \lccode'7='G
  \lccode'3='C \lccode'8='H
  \lccode'4='D \lccode'9='I
  \lccode'5='E \lccode'0='J
  \lowercase{\def\temp{#2}}%
  \expandafter\egroup\expandafter\def\csname#1\temp\endcsname{#3}}

\Name{test}{1}{Hello}
\Name{test}{2}{World!}
\testA\ \testB
```

Of course we don't need such a remapping at all, which is demonstrated by the alternative posted by Piet van Oostrum at TEX-NL:

```
\long\def\Name#1#2#3%
 {\expandafter\def\csname #1\ifcase#2\relax\or A\or B\or C\or
    D\or E\or F\or G\or H\or I\or J\or K\fi\endcsname{#3}}
```

A better variant is the following. This one takes care of the zero case:

```
\long\def\Name#1#2#3%
 {\expandafter\def\csname #1\ifcase#2 J\or A\or B\or C\or D\or
    E\or F\or G\or H\or I\else#2\fi\endcsname{#3}}
```

The next one is even better because it also takes care of non digits:

```
\long\def\Name#1#2#3%
 {\expandafter\def\csname#1\ifx#20J\else\ifcase0#2\or A\or B\or
    C\or D\or E\or F\or G\or H\or I\else#2\fi\fi\endcsname{#3}}
```

Now we can say:

```
\Name{test}{0}{zero}
\Name{test}{1}{one}
\Name{test}{x}{xxx}
\testJ \testA \testx
```

Bogusław Jackowski & Piotr Strzelczyk

Sigma tweak

A sum operator that adjusts its width automatically to the widest subscript or superscript – perhaps completely useless ;-)

Note: pdfTEX is required for clippling; you may wish to use other tools for this purpose
defining the parts of the symbol:

```
\def\SIGMAleft{\copy0 }
\def\SIGMAright{\copy1 }
\def\SIGMAcenter{\copy2 }
```

preparing the parts of the symbol:

```
\setbox0\hbox{\tenex \char88\kern-.6em}
\setbox1\hbox{\kern-.8em\tenex \char88}
\setbox2\hbox{\hbox{\kern-.8em\tenex \char88\kern-.6em}}
```

clipping:

```
\pdfxform0 \setbox0\hbox{\pdfrefxform\pdflastxform}
\pdfxform1 \setbox1\hbox{\pdfrefxform\pdflastxform}
\pdfxform2 \setbox2\hbox{\pdfrefxform\pdflastxform}
```

assembling:

```
\def\SIGMA{%
  \mathop{\hbox{\SIGMAleft \kern-.1em
  \xleaders\SIGMAcenter\hfill
  \kern-.1em\SIGMAright}}\limits}% with \nolimits behaves ''traditionally''
```

usage:

```
$ \SIGMA^{\rm a}\quad
  \SIGMA^{\rm sum}\quad
  \SIGMA^{\rm operator}\quad
  \SIGMA^{\rm that\ adjusts}\quad
  \SIGMA^{\rm its\ width\ automatically}
        _{\rm to\ the\ widest\ subscript\ or\ superscript}
$
```

As a result we get: