Hans Hagen
pragma@pi.net

Bijlage 30

Pretty printing TEX, MetaPost, Perl and JavaScript

Keywords
verbatim, MetaPost, Perl, JavaScript, CONTEXT

abstract
Although for real pretty printing of sources one has to use CWEB like environments, TEX can also do a pretty job rather well. The CONTEXT verbatim environment has pretty printing built in. One can either use colors of fonts. The latter is used in the MAPS, the former in this article.

A few years ago I implemented a verbatim environment that supports coloring of TEX. The source code can be found in `supp-ver.tex` and was presented in MAPS 16. Although primary meant for CONTEXT, this module is rather generic, which is proved by the fact that it is used for typesetting verbatim in the MAPS.

In this article I will introduce the successor of this module: `verb-ini.tex`. This new module is backward compatible, but offers a more general solution for pretty printing. The main enhancement is that the pretty printing interpreter is generalized and supports not only TEX, but also METAFONT and METAPOST, as well as PERL and JAVASCRIPT code. These filters are defined in the files `verb-*.tex`.

I needed the extensions because in CONTEXT metagraphics can be included in the document source. That way users can produce run time and layout dependant graphics. The PERL pretty printer was needed after I reimplemented TEXUTIL in PERL. Pretty printing of JAVASCRIPT came around when I wrote a calculator demo for the PDF platform that demonstrates how TEX, METAPOST and JAVASCRIPT have come together.

Before I go into detail, I'll show some examples of pretty printed code. First a TEX example.

```
\def\SomethingBoxed#1%
  {\framed[width=10cm,offset=.25cm]{#1}}

\SomethingBoxed{Something Boxed}
```

In METAPOST the visualization involves special treatment of reserved words. As one can see, both colors and fonts are used. Keywords like `btex` are handled according to their meaning and disable the interpretation of the following tokens until `etex` is met.

```
beginfig(1);
  draw fullcircle xscaled 200 yscaled 100;
  label(btex draw a {\em test} shape etex, (0,0));
endfig;
```

In TEXUTIL, the next piece of PERL deals with stripping unwanted characters from strings. Making strings healthy is needed for proper sorting of index entries.

```
sub SanitizedString
  { my ($string) = $_[0]; # my o my
```

```
    if ($ProcessQuotes)
      { $string =~ s/\\([\^\"\'\’\~\,])/$1/gio;
        $copied = $string;
        $copied =~ s/([\^\"\'\’\~\,])([a-zA-Z])/$ASCII{$1}/gio;
        $string =~ s/([\^\"\'\’\~\,])([a-zA-Z])/$2/gio;
        $string=$string.$copied }
    $string =~ s/\\-|\|\||/\-/gio;
    $string =~ s/\\[a-zA-Z]*|  |\{|\}//gio;
    return $string }
```

When defined, the PERL interpreter also understands special functions, like:

```
use Getopt::Long;
```

The JAVASCRIPT interpreter is implemented on top of the PERL one. The main difference lays in the way comments are handled: `//`, `/*` and `*/` versus `#`.

```
i = 1;
while (i<=100)
 { Stack[i] = ""; // We're talking about a stack of strings!
   i++ }          // This is in fact i = i+1;
if (Done)
  { Stack[1] = "in a while" }
else              /* such an else is optional */
  { Stack[1] = "at once" }
```

Due to the fact that the verbatim environment is sort of object oriented, each pretty printer get's it own commands:

```
\startTEX ··· \stopTEX
\startMP  ··· \stopMP
\startPL  ··· \stopPL
\startJV  ··· \stopJV
```

For plain TEX users these commands are available after loading the macros:

```
\input verb-ini
```

The interpreter is enabled by saying:

```
\setcolorverbatim
```

The current meaning of this macro takes care of POSTSCRIPT colors, but tuning the visualization to his or her personal needs, is not that hard.

In CONTEXT, users can not only use the `\start..` commands mentioned, but also adapt some characteristics of each individual verbatim environment, like:

```
\setuptyping[MP][margin=2em,space=on]
```

Of course one can use the commands `\starttyping`, `\typefile`, and `\typebuffer`. These obey the settings of the general typing environment, like:

```
\setuptyping[option=TEX,margin=1em]
```

The command `\typefile` uses the file extension to automatically determine the pretty interpreter to be used.

The colorization is implemented using the CONTEXT palet mechanism. This mechanism enables users to define collections of colors that can be switched as a whole. So rather than redefining a specific shade of red, yellow or whatever, one just enables ano-

ther palet that has flavors of them defined. The default colors are defined as:

```
\definecolor [colorprettyone]   [r=.9, g=.0, b=.0] % red
\definecolor [colorprettytwo]   [r=.0, g=.8, b=.0] % green
\definecolor [colorprettythree] [r=.0, g=.0, b=.9] % blue
\definecolor [colorprettyfour]  [r=.8, g=.8, b=.6] % yellow

\definecolor [grayprettyone]    [s=.30]
\definecolor [grayprettytwo]    [s=.45]
\definecolor [grayprettythree]  [s=.60]
\definecolor [grayprettyfour]   [s=.75]
```

There are two main palets, one for color and one for gray printing:

```
\definepalet
  [colorpretty]
  [  prettyone=colorprettyone,
     prettytwo=colorprettytwo,
   prettythree=colorprettythree,
    prettyfour=colorprettyfour]

\definepalet
  [graypretty]
  [  prettyone=grayprettyone,
     prettytwo=grayprettytwo,
   prettythree=grayprettythree,
    prettyfour=grayprettyfour]
```

These palets are inherited by the specific pretty palets, for instance:

```
\definepalet [MPcolorpretty]  [colorpretty]
\definepalet [MPgraypretty]   [graypretty]
```

By default we have:

```
\setuptyping[MP][palet=MPcolorpretty]
```

but when needed, one can specify another palet. Of course, such a palet should be defined first in terms of `prettyone` upto `prettyfour`.

In CONTEXT one could (and still can) make spaces visible, obey tabs and embed TEX commands (using a different escape character). New however is the way multiple empty lines and breaking paragraphs are handled. From now on, by default, multiple empty lines are concatenated into one. Also, by default, the first two and last two lines are always kept together.

Another new feature is dedicated to Kees van der Laan, who, as a MAPS author, would like to see the pretty printer adapt its interpretation to TEX's current active character state. The next piece of TEX code shows this feature:

```
\bgroup
\catcode`\|=\@@escape %%\|\
\catcode`\\=\@@active %%\\+
|gdef|dohandlenewpretty#1%
  {|def|dodohandlenewpretty##1%
     {|getprettydata{\}%
      |let|newprettytype=|prettytype
      |getprettydata{##1}%
      |ifnum|prettytype=|newprettytype
```

```
      |let|next=|newpretty
    |else
      |def|next{|newprettycommand{#1}##1}%
    |fi
    |next}%
  |def|donohandlenewpretty##1%
    {|newprettycommand{#1}##1}%
  |handlenextnextpretty
    |dodohandlenewpretty|donohandlenewpretty}
|egroup
```

Without the switch, the first few lines would look like:

```
\bgroup
\catcode`\|=\@@escape
\catcode`\\=\@@active
|gdef|dohandlenewpretty#1%
  {|def|dodohandlenewpretty##1%
     {|getprettydata{\}%
      |let|newprettytype=|prettytype
```

The redefinitions are invoked by the double comment sign, followed by a backslash. The next (non space) token will be interpreted as the one following it. In our example the | will be visualized as the \ and the \ as the + token.

When followed by a space, the double comment takes the next token as an interpreter command to be executed. An example demonstrates this feature.

```
\ziezo{test}          %%\ P   ##\ B##\ T % enter PERL mode       %%\ E
if $test eq "test"    ##\ B   ##\ B##\ T % begin group (\bgroup) %%\ E
if $test eq "test";   ##\ T   %%\ B%%\ T % enter TEX mode        %%\ E
\ziezo{test}          %%\ M   %%\ B%%\ T % enter METAPOST mode   %%\ E
draw (0,0)--(10,10);  %%\ E   ##\ B##\ T % end group (\egroup)   %%\ E
if $test eq "test";
```

this was typed in as (forget the comments):

```
\ziezo{test}          %%\ P
if $test eq "test"    ##\ B
if $test eq "test";   ##\ T
\ziezo{test}          %%\ M
draw (0,0)--(10,10);  %%\ E
if $test eq "test";
```

When in CONTEXT one wants to pass data from TEX to JAVASCRIPT, one can use the prefix TEX. This prefix is interpreted as *pretty print the next string as a TEX one*. Of course the keyword TEX is stripped before the JAVASCRIPT is shipped out. So:

```
var MinLevel    = -TEX \MinLevel;
var MaxLevel    = TEX \MaxLevel;
var Level       = 1;
```

becomes in pretty typography:

```
var MinLevel    = -TEX \MinLevel;
var MaxLevel    = TEX \MaxLevel;
var Level       = 1;
```