

# T<sub>E</sub>X – macro programming

Victor Eijkhout

Notes for CS 594 – Fall 2004

# The input processor

## From file to lines

- ▶ Lines lifted from file, minus line end
- ▶ Trailing spaces removed
- ▶ `\endlinechar` appended, if 0–255, default 13
- ▶ accessing all characters:  $\hat{n}$  with  $n < 128$  replaced by character  $\text{mod}(n + 64, 128)$ ; or  $\hat{\hat{xy}}$  with  $x, n$  lowercase hex replaced by character  $xy$ .

## Category codes

- ▶ Special characters are dynamic: character code to category code mapping during scanning of the line
- ▶ example: `\catcode36=3`, or `\catcode'\$=3`
- ▶ Assignment holds immediately!

Normal math `$n=1$`, `\catcode'\/=3 /x^2+y^2/`.

Output:

*Normal math  $n = 1, x^2 + y^2.$*

## Usual catcode assignments

- ▶ 0: escape character /, 1/2: beginning/end of group {},  
3: math shift \$, 4: alignment tab &, 5: line end, 6: parameter #
- ▶ 7/8: super/subscript ^\_, 9: ignored NULL
- ▶ 10: space, 11: letter, 12: other
- ▶ 13: active ~, 14: comment %, 15: invalid DEL

# Token building

- ▶ Backslash (really: escape character) plus letters (really: catcode 11)  $\Rightarrow$  control word, definable, many primitives given
- ▶ backslash plus space: control space (hardwired command)
- ▶ backslash plus any other character: control symbol; many default definitions, but programmable
- ▶  $\#n$  replaced by 'parameter token  $n$ ',  $\##$  replaced by macro parameter character
- ▶ Anything else: character token (character plus category)

## Some simple catcode tinkering

- ▶ `\catcode'\@=11`  
`\def\@InternalMacro{...}`  
`\def\UserMacro{ .... \@InternalMacro ... }`  
`\catcode'\@=12`

# States

- ▶ Every line starts in state  $N$
- ▶ in state  $N$ : spaces ignored, newline gives `\par`, with anything else go to  $M$  (middle of line)
- ▶ State  $S$  entered after control word, control space, or space in state  $M$ ; in this state ignore spaces and line ends
- ▶ State  $M$ : entered after almost everything. In state  $M$ , line end gives space token.

## How many levels down are we?

1. Lifting lines from file, appending EOL
2. translating  $\hat{\hat{xy}}$  to characters
3. catcode assignment
4. tokenization
5. state transitions

## What does this give us?

- ▶  $\text{T}_{\text{E}}\text{X}$  is now looking at a stream of tokens: mostly control sequences and characters
- ▶ Actions depend on the nature of the token: expandable tokens get expanded, assignments and such get executed, text and formulas go to output processing.
- ▶ Read chapters 1,2,3 of  $\text{T}_{\text{E}}\text{X}$  by Topic.

## Macros and expansion

# Expansion

- ▶ Expansion takes command, gives replacement text.
- ▶ Macros: replace command plus arguments by replacement text
- ▶ Conditionals: yield true or false branch
- ▶ Various tools
- ▶ Read chapters 11,12 of T<sub>E</sub>X by Topic.

# The basics of macro programming

# Macro definitions

- ▶ Simplest form: `\def\foo#1#2#3{ .. #1 ... }`
- ▶ Max 9 parameters, each one token or group:

```
\def\a#1#2{1:(#1) 2:(#2)}  
\a b{cde}
```

Output:

*1:(b) 2:(cde)*

## Delimited macro definitions

- ▶ Delimited macro arguments:

```
\def\a#1 {Arg: '#1'}  
\a stuff stuff
```

Output:

*Arg: 'stuff'stuff*

- ▶ Delimited and undelimited:

```
\def\Q#1#2?#3!{Question #1: #2?\par Answer: #3.}  
\Q {5.2}Why did the chicken cross  
the Moebius strip?Eh\dots!
```

Output:

*Question 5.2: Why did the chicken cross the  
Moebius strip?*

*Answer: Eh....*

## Grouping

- ▶ Groups induced by  
`{}` `\bgroup` `\egroup` `\begingroup` `\endgroup`
- ▶ `\bgroup`, `\egroup` can sometimes replace `{}`
- ▶ `\begingroup`, `\endgroup` independent
- ▶ funky stuff:

```
\def\open{\begingroup} \def\close{\endgroup}  
\open ... \close
```

```
\newenvironment{mybox}{\hbox\bgroup}{\egroup}  
A \begin{mybox}B\end{mybox} C
```

Output:

*A B C*

- ▶ Chapter 10 of T<sub>E</sub>X by Topic.

## More tools

- ▶ Counters:

```
\newcount\MyCounter \MyCounter=12  
\advance\MyCounter by -3 \number\MyCounter  
also \multiply, \divide
```

- ▶ Test numbers by

```
\ifnum\MyCounter<3 <then part>\else <else part> \fi  
available relations: > < =; also \ifodd, and  
\ifcase\MyCounter <case 0>\or <case 1> ...  
  \else <other> \fi
```

Only a finite number of counters in T<sub>E</sub>X;  
use `\def\Constant{42}` instead of

```
\newcount\Constant \Constant=24
```

# Skips

- ▶ (technically: glue)
- ▶ Finite: `\vskip -3pt \hskip 10pt plus 1cm minus 1em`
- ▶ Infinite: `\hfil, \hfill, \vfil, \vfill`
- ▶ Both ways: `\hss` is `\hskip 0pt plus 1fill minus 1fill`
- ▶ There are lots of built-in skip parameters

- ▶ User defined: `\newdimen\MySize`, `\newskip\MyGlue`
- ▶ Assignment, `\multiply`, `\divide`, `\advance`

# Conditionals

- ▶ General form `\if<something> ... \else ... \fi`
- ▶ Already mentioned `\ifnum`, `\ifcase`  
`\ifnum\value{section}<3` Just getting started.  
`\else On our way\fi`  
Output:  
*Just getting started.*
- ▶ Chapter 13 of T<sub>E</sub>X by Topic

- ▶ Programming tools: `\iftrue`, `\iffalse`  
`\iftrue {\else }\fi \iffalse {\else }\fi`
- ▶ `\ifx` equality of character (char code and cat code); equality of macro definition
- ▶ `\if` equality of character code after expansion.

## Bunch of examples

## Grouping trickery

Bad idea:

```
\def\parbox#1#2{%  
  \vbox{\setlength{\textwidth}{#1}{#2}}}
```

Better:

```
\def\parbox#1{%  
  \vbox\bgroup \setlength{\textwidth}{#1}  
  \let\next=}
```

Then `\parbox{3in}{ <bunch of text> }` becomes

```
\vbox\bgroup  
  \setlength{\textwidth}{3in}  
  \let\next={ <bunch of text> }
```

## Use of delimited arguments

```
\def\FakeSC#1#2 {%  
  {\uppercase{#1}\footnotesize\uppercase{#2}\ }%  
  \FakeSC}
```

Then `\FakeSC word` gives #1  $\leftarrow$  w , #2  $\leftarrow$  ord.

Expansion of the macro invocation `\FakeSC word` gives

```
{\uppercase{w}\footnotesize\uppercase{ord}\ }\FakeSC
```

```
\FakeSC This sentence is fake small-caps .
```

Output:

*THIS SENTENCE IS FAKE SMALL-CAPS .*

## How did I stop that recursion?

```
\def\periodstop{.}
\def\FakeSC#1#2 {\def\tmp{#1}%
  \ifx\tmp\periodstop
    \def\next{.}
  \else
    \def\next{%
      {\uppercase{#1}\footnotesize\uppercase{#2}\ }%
      \FakeSC}%
  \fi \next}
```

## explanation of `\FakeSC`

- ▶ Invocation `\FakeSC word` gives `\def\tmp{w}` so `\ifx` is false
- ▶ `\else` case does fake small-caps and recursive call
- ▶ Invocation `\FakeSC .` gives `#1 <- .` and `#2` is empty.
- ▶ `\ifx` test is now true, definition of next reinserts period

## Two-step macros

- ▶ Wanted:

```
\PickToEOL This text is the macro argument  
and this is not
```

- ▶ Basic idea: argument delimited by line end

```
\def\PickToEOL#1^^M{ <stuff> }
```

- ▶ Wrong:  $\TeX$  stops processing at  $^^M$

## attempt #1

- ▶ Change catcode of line end

```
\def\PickToEOL
  {\begingroup\catcode'\^^M=12 \xPickToEOL}
\def\xPickToEOL#1^^M{ ...#1... \endgroup\par}
```

- ▶ Invocation:

```
\PickToEOL line of text
=>
```

```
\begingroup\catcode'\^^M=12 \xPickToEOL line of text
```

- ▶ Invocation is correct;  
\xPickToEOL definition is still not right

## attempt #2

- ▶ Conditions at the definition count

```
\def\PickToEOL
  {\begingroup\catcode'\^^M=12 \xPickToEOL}
{\catcode'\^^M=12 %
  \gdef\xPickToEOL#1^^M{ \textbf{#1}\endgroup\par}
}
\PickToEOL This text is the macro argument
and this is not
```

Output:

```
fl This text is the macro argument
and this is not
```

- ▶ `\gdef` is 'global' define, needed because of group

## Optional arguments

- ▶ Example: `\section[Short]{Long title}`

- ▶ Need two macros:

```
\def\sectionwithopt[#1]{#2}{ <stuff> }  
\def\sectionnoopt#1{\sectionwithopt[#1]{#1}}
```

and way to choose between them

- ▶ Wrong way:

```
\def\brack{[]}  
\def\section#1{\def\tmp{#1}  
  \ifx\tmp\brack  
  % this will never work
```

use of `\futurelet`

```
\let\brack[
\def\section{\futurelet\next\xsection}
\def\xsection
  {\ifx\next\brack
    \let\next\sectionwithopt
  \else \let\next\sectionnoopt \fi \next}
\def\sectionnoopt#1{\sectionwithopt[#1]{#1}}
\def\sectionwithopt[#1]#2{Arg: '#2'; Opt '#1'}
\section[short]{Long}\par
\section{One}
```

Output:

*Arg: 'Long'; Opt 'short'*

*Arg: 'One'; Opt 'One'*

## Expanding out of sequence

- ▶ Suppose

```
\def\a#1#2{Arg1: #1, arg2: #2.}
```

```
\def\b{{one}}{two}}
```

How do you give the contents of `\b` to `\a`?

- ▶ Wrong: `\a\b`

Solution: `\expandafter\a\b` expands `\b`, then `\a`

- ▶ Suppose `\def\c{\b}`, how would you get `\a\c` to work?

## More expansion trickery

- ▶ The  $\LaTeX$  command `\newcounter{my}` executes a command `\newcount\mycounter`. How is that name formed?
- ▶ Form control sequence names with `\csname stuff\endcsname`. However `\def\newcounter#1{\newcount\csname #1counter\endcsname}` would define a counter name `\csname`.
- ▶ We need to activate `\csname` before `\newcount`

## solution with `\expandafter`

- ▶ Sequence `\expandafter\ a\ b` expands `\ b`, then `\ a`
- ▶ Improved definition

```
\def\newcounter#1{%  
    \expandafter\newcount\csname #1counter\endcsname}
```

- ▶ Use

```
\newcounter{my} =>  
\expandafter\newcount\csname mycounter\endcsname =>  
    \newcount\mycounter
```

## Expanding definition

- ▶ `\edef\foo{ . . . . }` first expands the body, before doing the definition.
- ▶ Use as tool (above example revisited)

```
\def\a#1#2{Arg1: #1, arg2: #2.}  
\def\b{{one}{two}}  
\edef\tmp{\noexpand\a\b} % <= expansion of \b  
\tmp                       % <= call of \a
```

more with `\edef`

- ▶ Remember the catcode trickery:

```
\edef\restoreatcat{\catcode'\@=\the\catcode'\@\relax}  
\catcode'\@=11  
\def\@foo{...}  
\restoreatcat
```

- ▶ Problem: restoring catcode correctly
- ▶ Defining based on current conditions:

```
\edef\restoreatcat{\catcode'\@=\the\catcode'\@}  
and use \restorecat instead of \catcode'\@=12
```

- ▶ If catcode of @ is 4, then `\the\catcode'\@` expands to 4, so the `\edef` is equivalent to  

```
\def\restorecat{\catcode'\@=4 }
```

## All together now:

- ▶ Ponder this:

```
\edef\foo
  {\expandafter\noexpand\csname bar\endcsname}
```

- ▶ Call `\foo` becomes

```
\expandafter\noexpand\csname bar\endcsname
  \noexpand\bar
    \bar
```

## Nested macro definitions

- ▶ Wrong: `\def\ a{\def\ b#1{}}`  
error message that `\ a` does not have argument
- ▶ Also wrong: `\def\ a#1{\def\ b#1{}}`  
`\ a ? %` becomes  
`\def\ b?{}`  
so `\ b` is a macro that has to be followed by `?`.

## nested definitions, solution

- ▶ Remember that ## is replaced by #:  
`\def\aa#1{\def\bbb#1#1{Arg: '##1'}}}`
- ▶ Now `\aa ?` becomes `\def\bbb#1?{}`,  
macro `\bbb` has one argument, delimited by `?`  
(basic idea for `\verb` macro)

- ▶ Test

```
\aa ! \bbb word words!\par  
\aa s \bbb word words!
```

Output:

```
Arg: 'word words'  
Arg: 'word word'!
```

## To summarize your toolbox

- ▶ `\def`, `\edef`
- ▶ `\expandafter`, `\noexpand`
- ▶ `\csname`, `\endcsname`
- ▶ `\let`, `\futurelet`

## How do you debug this stuff?

- ▶ The  $\text{T}_{\text{E}}\text{X}$  equivalent of `printf...`
- ▶ `\message`
- ▶ `\tracingmacros=2`
- ▶ output goes into the log file