WRITING MACROS WITH TEXAPI

Author: Paul Isambert
Version: 1.0
Date: 6/17/2010

Typeset in Chaparral Pro (Carol Twombly) and Lucida Sans Typewriter (Charles Bigelow and Kris Holmes) with LuaTEX v.60.1.

The first motivation for this set of macros is selfish: after rewriting the same lines over and over and wasting so many excruciating (yes!) hours debugging intricate loops with one typo, I decided I could use a toolkit containing the painful code without errors (hopefully) and use it for future packages.

The second motivation is more ambitious: I think it's a pity so many packages are written for one format and are thus unusable outside it, even though those packages could be useful to anybody. This is so, I believe, because a format mixes two different things: decisions about typesetting (mainly) and utility macros. The former are the essence of a format, whereas the latter are just shorthands you can use or not, replace, or ignore completely. But the fact is that users of a format tend to use the utility macros shipped with it, and thus writes macros that can't be reused elsewhere, even though nothing crucial hinges on what utility macros one uses. Thus, `texapi` aims at providing a good deal of this kind of macros whitout relying on any particular format, so that one can write code without having to take into account how it will be used. Moreover, `texapi` is also format-aware, meaning some commands are defined differently depending on the format being used, and one doesn't have to create as many macros as there are formats.

There is at least one basic assumptions, namely that formats should contain plain TEX's allocation macros. This the case for all formats I know.

In what follows, commands have a grey background when they are fully expandable, e.g. they can be used inside `\csname`…`\endcsname`, provided you don't use them with unexpandable arguments, of course. On the other hand, all unexpandable commands are protected.

Arguments are denoted by *<text>*, where 'text' makes the intended use clearer and doesn't denote any particular type of argument, except in the case of *<command>*, which denotes a control sequence (something expandable, actually), and *<csname>*, which denotes an argument suitable to `\csname`. Braces are indicated only when mandatory, but of course they can be used to delimit arguments as usual.

ENGINE AND FORMAT DETECTION

`\texenginenumber` This is a `\chardef`'ined number set according to the engine used: 0 means e-TEX, or an unknown engine with e-TEX extensions; 1 means XeTEX (detected because `\XeTeXinterchartoks` exists); 2 means pdfTEX (detected thanks to `\pdfstrcmp`); 3 means LuaTEX (detected thanks to `\directlua`). Numbering here allows one to detect a pdfTEX-based engine with `\texenginenumber>1`. ConTEXt has an equivalent `\texengine`, with pdfTEX=1 and XeTEX=2, though.

`\formatnumber` This number does the same with formats. Here, 0 means an unknown format, 1 means plain (because `\fmtname` is '`plain`'), 2 means eplain (because `\fmtname` is '`eplain`'), 3 means ConTEXt (because there exists an `\inspectnextoptionalcharacter` command), 4 means LaTEX2e (because `\fmtname` is '`LaTeX2e`') and 5 means LaTEX3 (because there exists an `\ExplSyntaxOn` command). There's no distinction yet between LaTEX3 on top of LaTEX2e and LaTEX3 as a format per se. Since `\formatnumber` is set only if it doesn't already exist, you can write your package with, say, `code.tex` containing the main code and `code.sty` and `t-code.tex` as wrapper files for LaTEX and ConTEXt respectively, with `\formatnumber` already set accordingly.

`\priminput` `\primunexpanded` Both LaTEX and ConTEXt redefines `\input`, and in ConTEXt `\unexpanded` has not the meaning of the e-TEX primitive. These two commands are thus the primitive `\input` and `\unexpanded` respectively.

`\loadmacrofile<file>` The behavior of this command depends on `\formatnumber`. The *<file>* should be given without extension, and the following happens: in ConTEXt, `\usemodule[`*<file>*`]` is executed, in LaTEX `\RequirePackage{`*<file>*`}` is used, and in other formats it is simply `\input`*<file>*`.tex`. This makes sense only with packages that are distributed as described above, i.e. with the main code in one file and wrapper files for LaTEX and ConTEXt, like Ti*k*Z or *librarian*.

`\senderror<package><message>` This sends an error message according to the format's custom. In `plain` and `eplain` (and in an unknown format), it produces `\errmessage{`*<package>* `error: `*<message>*`}`. In LaTEX, we get `\PackageError{`*<package>*`}{`*<message>*`}{}` (no help message) and in ConTEXt `\writestatus{`*<package>* `error}{`*<message>*`}` (which is far from ConTEXt's sophisticated communication system but, well...).

```
\def\myengine{%
  \ifcase\texenginenumber
    e-\or Xe\or pdf\or Lua\fi\TeX
  }
\def\myformat{%
  \ifcase\formatnumber
    unknown\or plain\or eplain\or
    ConTeXt\or LaTeX\or LaTeX3\fi
  }
This documentation has been typeset
with \myformat\ under \myengine.
```

This documentation has been typeset with plain under LuaTEX.

ARGUMENT MANIPULATION

`\emptycs`
`\spacecs`
`\spacechar`

Pretty useful macros whose meaning is clear, but whatever: `\emptycs` is an emptily defined command, `\spacecs` expands to a space, and `\spacechar` denotes a space, i.e. it is an implicit space and not really a macro.

```
This is \gobbletwoand{very } uninteresting.
```

This is very interesting.

`\gobbleone`
`\gobbleoneand`*<code>*

Those, as you might imagine, gobble the following argument; the second version also excutes *<code>* afterwards. There are actually nine such commands in each case, and they are (for the sake of completeness) `\gobbleone`, `\gobbletwo`, `\gobblethree`, `\gobblefour`, `\gobblefive`, `\gobblesix`, `\gobbleseven`, `\gobbleeight` (watch out, two *e*'s) and `\gobblenine` for the first version, and `\gobbleoneand`, `\gobbletwoand`, `\gobblethreeand`, `\gobblefourand`, `\gobblefiveand`, `\gobblesixand`, `\gobblesevenand`, `\gobbleeightand` and `\gobblenineand`. Note that `\gobblenineand`*<code>* takes two expansion steps to return *<code>*, instead of only one in the other cases.

```
\def\foo#1#2{\detokenize{(1=#1,2=#2)}}
\def\bar{two}
\foo{one}\bar
\passexpanded{\foo{one}}\bar
```

(1=one,2=\bar)(1=one,2=two)

`\unbrace`*<code>*

This is the kind of command you probably can't see the point of until you need it. It returns its *<code>* untouched, but with outermost braces removed if any.

`\swapargs` *<arg1><arg2>*
`\swapbraced` *<arg1><arg2>*
`\swapleftbraced` *<arg1><arg2>*
`\swaprightbraced`*<arg1><arg2>*

The first of those returns *<arg2><arg1>* into the stream, without any brace to delimit them. On the contrary, `\swapbraced` returns {*<arg2>*}{*<arg1>*}. And, as you might imagine, `\swapleftbraced` returns {*<arg2>*}*<arg1>* whereas `\swaprightbraced` returns *<arg2>*{*<arg1>*}.

`\passexpanded` *<arg1><arg2>*
`\passexpandednobraces`*<arg1><arg2>*

The first one returns *<arg1>*{*<arg2 expanded once>*} and the second *<arg1><arg2 expanded once>*. It's some sort of long `\expandafter` built on `\swapargs` and associates, and if *<arg1>* is a single token it's faster to use `\expandafter` itself. It's not a real `\expandafter`, though, since *<arg2>* is expanded to the left of *<arg1>* and then moved back to its right. Which, with e.g. an `\else` as *<arg2>*, will lead to results you probably haven't foreseen and expected. If *<arg2>* is some material you want to turn into a command with `\csname`, see `\passcs` below.

DEFINING & USING COMMANDS

`\defcs` *<csname><parameter text>{<definition>}*
`\edefcs` *<csname><parameter text>{<definition>}*
`\gdefcs` *<csname><parameter text>{<definition>}*
`\xdefcs` *<csname><parameter text>{<definition>}*

These work exactly like `\def`, `\edef`, `\gdef` and `\xdef`, except they define a command with name *<csname>*. The *<parameter text>* is the usual one, and any the space at the beginning is significant. I.e. `\defcs{foo}#1{...}` and `\defcs{foo} #1{...}` aren't equivalent at all. Prefixes can be appended as with `\def`.

```
\defcs{foo}#1{This is foo: #1.}
\foo{bar}
```

This is foo: bar.

`\letcs` *<csname><command>*
`\lettocs` *<command><csname>*
`\letcstocs` *<csname><csname>*

These `\let` the first command or command named *<csname>* to the meaning of the second one. In both `\lettocs` and `\letcstocs`, if the command with name *<csname>* is undefined, it is not let to `\relax`. So these are different from `\let` with `\expandafter`'s. The `\letcs` command can also be used to create an implicit character, of course.

```
\expandafter\let\expandafter\foo
                \csname undefined\endcsname
\lettocs\bar{reallyundefined}
\letcstocs{reallyundefined}{reallyundefined}

Compare this: \meaning\foo,
and that: \meaning\bar.
And better yet: \meaning\reallyundefined.
```

Compare this: \relax, and that: undefined. And better yet: undefined.

`\addleft` *<command><material>*
`\addleftcs` *<csname><material>*
`\eaddleft` *<command><material>*
`\eaddleftcs` *<csname><material>*

This redefines *<command>* or a command named *<csname>* to itself with *<material>* added at the beginning. The e-variant performs an `\edef` so that *<material>* is fully expanded (but not *<command>*). The usual prefixes can be appended.

`\addright` *<command><material>*
`\addrightcs` *<csname><material>*
`\eaddright` *<command><material>*
`\eaddrightcs` *<csname><material>*

This is the same thing as above, but the material is added at the end. In both the left and right version, the command thus redefined should be a simple command working by itself (i.e. no argument and no delimiter). In the *<csname>* case, no check is performed to ensure that *<csname>* is defined (but in the worst case it ends up as `\relax`, because of its being called after the implicit `\def` (get it?)).

```
\defcs{foo}{bar}
\addleftcs{foo}{In a }
\addright\foo{ (how fascinating).}
\foo
```

In a bar (how fascinating).

`\usecs` *<csname>*
`\usecsafter` *<csname>*
`\passcs` *<code><csname>*
`\passexpandedcs` *<code><csname>*
`\noexpandcs` *<csname>*
`\unexpandedcs` *<csname>*

Various ways to use a command with name *<csname>*: `\usecs` performs a simple `\csname`*<csname>*`\endcsname` (and doesn't even check whether *<csname>* is defined or not, so this might relax it a little bit), `\usecsafter` does the equivalent of `\expandafter\command`, `\passcs` puts *<csname>* as a real (unbraced) command after *<code>*, whereas `\passexpandedcs` passes the expansion of the control sequence with name *<csname>* to *<code>*; `\noexpandcs` and `\unexpandedcs` return *<csname>* with a `\noexpand` prefix or its expansion as argument to `\unexpanded` (`\primunexpanded`, really).

```
\def\bar{whatever} \def\foo#1{[#1]}
I use it: \usecs{bar},
I use it after: \usecsafter{foo}\bar,
and I pass it: \passcs\foo{bar}.
```

I use it: whatever, I use it after: [w]hatever, and I pass it: [whatever].

`\commandtoname` *<command>*

This returns the name of *<command>*, i.e. *<command>* without its backslash (and made of catcode-12 characters, since it's based on `\string`).

```
\def\foo{\bar}
I don't expand it:
\edef\foobar{\noexpandcs{foo}}%
\meaning\foobar.
Or just a little bit:
\edef\foobar{\unexpandedcs{foo}}%
\meaning\foobar.
```

I don't expand it: macro:->\foo . Or just a little bit: macro:->\bar .

TESTS WITH COMMANDS

`\reverse`

Conditionals in `texapi` (not only those on page) can be prefixed with `\reverse`, so that if they're true the *<false>* argument is executed (if specified), and if they're false, the *<true>* argument is executed. So this is equivalent to `\unless`.

```
\ifcommand\TeX{Cool}{Too bad}.
Nothing: \iffcommand\undefined{Whatever}.
```

Cool. Nothing: .

`\ifcommand` *<command><true><false>*
`\iffcommand`*<command><true>*

This conditional executes *<true>* if *<command>* is defined. So it is a straight version of `\ifdefined`. The `\iff...` version, like all `texapi`'s `\iff...`, considers only the *<true>* case (which becomes the *<false>* case if the conditional is prefixed with `\reverse`).

```
\reverse\iffcs{undefined}{This command is undefined.}
```

This command is undefined.

`\ifcs` *<csname><true><false>*
`\iffcs`*<csname><true>*

Same as above, but with an `\ifcsname` this time. It goes without saying that *<csname>* isn't let to `\relax` thereafter if it was undefined.

```
\def\foo{}
\ifemptycommand\foo{Empty}{Not empty}.
\reverse\iffemptycs{gobbleone}{It ain't empty}.
```

Empty. It ain't empty.

`\ifemptycommand` *<command><true><false>*
`\iffemptycommand`*<command><true>*

This is true if *<command>* is defined with an empty definition text, i.e. it is equivalent to `\emptycs`. This is not true if *<command>* takes arguments, though, so `\gobbleone` isn't empty in this sense.

```
\iffxcs{undefined}\undefinedtoo
       {Same definitions.}
\ifxcscs{foo}{TeX}
        {These are the same}
        {These are different}.
```

Same definitions. These are different.

`\ifemptycs` *<csname><true><false>*
`\iffemptycs`*<csname><true>*

Same as above with a command named *<csname>*.

`\ifxcs` *<csname><command><true><false>*
`\iffxcs`*<csname><command><true>*

This is true if *<csname>* has the same definition as *<command>*, or they're both undefined.

`\ifxcscs` *<csname><csname><true><false>*
`\iffxcscs`*<csname><csname><true>*

This is true if both *<csname>*'s have the same definition, or they're both undefined.

VARIOUS CONDITIONALS

`\newife<command>`

This defines a conditional like plain TEX's `\newif`, except it takes two arguments (the *<true>* and *<false>* values) instead of an `\else`... `\fi` structure. Besides, this conditional is reversible with `\reverse`, and a 'double-f' (i.e. `\iff`...) version is also created, which takes the *<true>* part only. As with `\newif`, *<command>* must begin with `if`. (The *e* means *expandable*, although there's nothing more expandable in the conditionals thus constructed than in those defined with `\newif`, but anyway.)

```
\newife\iffoo
\iffoo{There is foo}{There is no foo}.
\footrue
\reverse\iffoo{There is no foo}{There is foo}.
\ifffoo{With three f's in a row}.
```

There is no foo. There is foo. With three f's in a row.

`\straightenif <TEX conditional><arg><true><false>`
`\straighteniff<TEX conditional><arg><true>`

Apart from `\ifdefined` and `\ifcsname` (in the guise of `\ifcommand` and `\ifcs` respectively), none of TEX's primitive conditionals are redefined in a straight fashion, i.e. with two arguments instead of `\else`... `\fi`. These commands let you use TEX's conditional in such a way. *<TEX conditional>* means such a primitive without a backslash (so this construction can be used inside real conditionals), e.g. `ifnum` or `ifvoid`. The *<arg>* is whatever you normally feed to this conditional. It is brutally concatenated, and you're the one in charge of adding space if needed, as for instance with `ifnum`. Chaos will ensue if you fail to do so. With conditionals that don't require anything, e.g. `iftrue` or `ifvmode`, leave *<arg>* empty (but don't forget it). Finally, *<true>* and *<false>* are executed accordingly, and the whole macro can be prefixed with `\reverse`.

```
                        % See this space?
\straightenif{ifnum}{1=1 }{Reality is preserved}
                           {Bad news}.
\reverse\straighteniff{if}{ab}
             {Different letters, obviously.}
\straightenif{iftrue}{}{Good}{Bad}.
```

Reality is preserved. Different letters, obviously. Good.

`\afterfi <code>`
`\afterdummyfi<code>`

You shouldn't use these. The first one closes the current conditional and executes *<code>*. The second one lets go one `\fi` and executes *<code>*. So these are kinds of `\expandafter`'s when *<code>* isn't just a command. Anything before the incoming `\fi` is gobbled. The reason why you should use one or the other should be clear to you, otherwise you'll probably be messing with a conditional.

```
\iftrue
  \afterdummyfi{\afterfi{Here we are.}}
\else
  \iffalse
    Whatever.
  \fi
\fi
```

Here we are.

POKING AT WHAT COMES NEXT

`\nospace<code>`

This gobbles any incoming space, if any, and executes *<code>*. Of course it doesn't require there to be any space to work properly.

```
\nospace{foo} bar
```

foobar

All the following conditionals can be prefixed with `\reverse`. And in case your head's buzzing, their names are quite regular: take an `\if`, `\ifcat` or `\ifx`, add '`next`', and create variants by doubling the `f` and/or adding `nospace` at the end.

```
Here comes \ifnext e{an }{a }e.
Here comes \reverse\ifnext e{a }{an }b.
Here comes
\iffnextnospace\foo{a control sequence: } \TeX.
```

Here comes an e. Here comes a b. Here comes a control sequence: TEX.

`\ifnext` *<token><true><false>*
`\iffnext` *<token><true>*
`\ifnextnospace` *<token><true><false>*
`\iffnextnospace` *<token><true>*

These poke at the next token and see whether it has the same character code as *<token>*. In other words, an `\if` test is performed between *<token>* and the next token in the input stream. However, neither *<token>* nor the incoming token are expanded, so that they can be control sequences and no unwanted expansion will occur. Control sequences are all equal according to this test (which can very well take an undefined control sequence as *<token>*). The nospace version must be pretty clear: the macro discards all incoming spaces until it finds a non-space token to test (unfortunately, an implicit space and a space character are undistinguishable as far as this test (and the next ones) is concerned, so in the very unlikely case where an implicit space was waiting in the stream, it'll be gobbled in the nospace variant).

```
\def\tex{\TeX\iffcatnext a{ }}
A \tex is a \tex is a \tex.
```

A TEX is a TEX is a TEX.

```
\def\foo{not \string\TeX}
\reverse\iffxnextnospace\TeX
  {The incoming command isn't \string\TeX: } \foo.
```

The incoming command isn't \TeX: not \TeX.

`\ifcatnext` *<token><true><false>*
`\iffcatnext` *<token><true>*
`\ifcatnextnospace` *<token><true><false>*
`\iffcatnextnospace` *<token><true>*

These are the same as above with an `\ifcat` test instead of `\if`. Again, control sequences aren't expanded and they all have the same category code.

`\ifxnext` *<token><true><false>*
`\iffxnext` *<token><true>*
`\ifxnextnospace` *<token><true><false>*
`\iffxnextnospace` *<token><true>*

Once again like the previous commands, this time with an `\ifx`, i.e. the definitions of control sequences are compared, and in case *<token>* and/or the next token are unexpandable thing, both character code and category code are compared. So these are performing real `\ifx` tests.

STRING MANIPULATION

`\ifstring` *<string1><string2><true><false>*
`\iffstring`*<string1><string2><true>*

These return *<true>* if the two strings are identical. Category codes aren't taken into account when strings are compared.

```
Two \ifstring{abc}{abc}{equal}{unequal} strings
and an \reverse\iffemptystring{something}{unempty} one.
```

Two equal strings and an unempty one.

`\ifemptystring` *<string><true><false>*
`\iffemptystring`*<string><true>*

These return *<true>* if *<string>* is empty.

`\newstring`*<string>*

The following operations (`\ifprefix`, `\removesuffix`, etc.) aren't fully expandable by default. However, if a string has been previously declared with `\newstring`, they magically become fully expandable.

So, in what follows, macros aren't marked as expandable, although they can be if the preceding condition is fulfilled. Besides, these macro aren't `\protected` even though their default behavior would require that they be. But you can always append a `\noexpand` to an unprotected command, whereas you cannot force the execution of a protected one. (This protecting issue is of course totally irrelevant for the `\removeprefixin` and `\removesuffixin` commands, which aren't expandable by definition and are thus protected.)

```
\newstring{abc}
\edef\foo{\ifprefix{abc}{abcd}{True}{False}.}
\edef\bar{\reverse\iffsuffix{abc}{whatever}{No suffix}.}
\edef\foobar{\ifcontains{abc}{gee}{Yes}{No}.}
\meaning\foo\par
\meaning\bar\par
\meaning\foobar
```

macro:->True.
macro:->No suffix.
macro:->No.

`\ifprefix` *<prefix><string><true><false>*
`\iffprefix`*<prefix><string><true>*

This test is true if *<string>* begins with *<prefix>*. Category codes do matter.

`\ifsuffix` *<suffix><string><true><false>*
`\iffsuffix`*<suffix><string><true>*

True if *<string>* ends with *<suffix>*.

`\ifcontains` *<string1><string2><true><false>*
`\iffcontains`*<string1><string2><true>*

Finally, this is true if *<string2>* contains *<string1>*.

```
\def\record#1 : #2.{%
  \par\bgroup
    \it\ifprefix*{#1}{\removeprefix*{#1} [live]}{#1}
  \egroup
  (\ifcontains/{#2}{\splitstring/{#2}{\dodate}}{#2})
  }
\def\dodate#1#2{recorded #1, released #2}

A somewhat incomplete list of fantastic
records by Frank Zappa:
\record Absolutely Free : 1967.
\record The Grand Wazoo : 1972.
\record L\"ather : 1977/1996.
\record *Make a Jazz Noise Here : 1988/1991.
```

A somewhat incomplete list of fantastic records by Frank Zappa:
Absolutely Free *(1967)*
The Grand Wazoo *(1972)*
Lather *(recorded 1977, released 1996)*
Make a Jazz Noise Here [live] *(recorded 1988, released 1991)*

`\removeprefix`*<prefix><string>*
`\removesuffix`*<suffix><string>*

These return *<string>* without *<prefix>* (resp. *<suffix>*). No test is performed to check that *<string>* indeed begins (resp. ends) with *<prefix>* (resp *<suffix>*), so these macros make sense only after the adequate tests.

`\removeprefixand`*<prefix><string><code>*
`\removesuffixand`*<suffix><string><code>*

These do the same as the previous one, but feed the resulting string to *<code>*, between braces. Once again, no test is performed beforehand.

`\removeprefixin`*<prefix><string><command>*
`\removesuffixin`*<suffix><string><command>*

These define *<command>* as *<string>* without *<prefix>* (resp. *<suffix>*). No test either. Sorry.

`\splitstring`*<string1><string2><code>*

This cuts *<string2>* in two at *<string1>*'s first occurrence and passes the two parts as braced arguments to *<code>*. And, again: no test.

VARIOUS THINGS ON THE SAME PAGE

`\setcatcodes{<list>}` The *<list>* argument here means comma separated *<characters>=<category code>*, with an *s* to *characters* because you can concatenate them if you want them to share the same *<category code>*. So, as you might have guessed, this set all *<characters>* to characters with catcode *<category code>*. And it also sets `\restorecatcodes` accordingly. The changes are local. The # character requires a backslash (so do braces and the backslash itself, but that's obvious).

```
\setcatcodes{\\\#\{\}\%=12,\|=0}
Hey, were're verbatimizing:
\def\foo#1{\bar{#1}}%
|restorecatcodes

Hey, were're verbatimizing: \def\foo#1{\bar{#1}}%
```

`\resetcatcodes` This restores the catcodes of the characters changed with the previous command, which is cumulative, i.e. `\restorecatcodes` restores catcodes changed by all preceding `\setcatcodes` commands, not only the last one. Since changes are local, `\restorecatcodes` may be useless in a group (and the effect of `\restorecatcodes` itself is local too).

```
\bgroup
And now in a group:\par
\setcatcodes{z=13}
\defz{ZZZZZZZZZZZZZZZZZZZZZZZZZ}
I'm sleeping: z.\par
\egroup
And I'm not: z.

And now in a group:
I'm sleeping: ZZZZZZZZZZZZZZZZZZZZZZZZZ.
And I'm not: z.
```

The trimming macros below are adapted from Will Robertson's `trimspace` package.

`\trimleft <string>`
`\trimright<string>`
`\trim <string>`

These return *<string>* with one space removed at the beginning or end or both. There's no need to check beforehand whether there are indeed such spaces.

```
+\trim{ bar }+

+bar+
```

`\passtrimleft <string><code>`
`\passtrimright<string><code>`
`\passtrim <string><code>`

These return *<string>* trimmed of spaces as a braced argument to *<code>*.

```
\deftrimleft\foo{ bar }
+\foo+

+bar +
```

`\deftrimleft <command><string>`
`\deftrimright<command><string>`
`\deftrim <command><string>`

The same thing again, except now those commands define *<command>* to *<string>*, etc.

WHILE STATEMENTS

`\repeatuntil`*<number><code>*

This executes *<code>* *<number>* times. The *<number>* argument can be a `\count` register, an integer defined with `\chardef`, etc., and of course a string of digits. In any case, it is really an argument and must be surrounded by braces if it is made of more than one token.

```
We have seen \repeatuntil\pageno{I} pages.\par
\edef\foo{\repeatuntil3{.}}
\meaning\foo
```

We have seen IIIIIIIII pages.
macro:->...

`\dowhile`*<condition><code>*

This repeats *<code>* while *<condition>* is true. The latter must be a 'straight' `if`, i.e. either one of `texapi`'s `\if...` or a `\straightenif{`*<TEX conditional>*`}` construction, in both cases without the *<true>* and *<false>* arguments, because *<true>* is actually *<code>*, and *<false>* would make no sense. Finally, the conditional must be a simple `\if...`, not an `\iff...` version. Once again, this makes sense: the *if and only if* clause is implicit in a *while* statement. If you use an `\iff...`, you'll end up with many empty braces, which is harmless unless you're in a context of expansion. You can use `\reverse` in *<condition>*.

```
\newife\ifbreakloop \def\foo{}
\dowhile{\reverse\ifbreakloop}
        {\addleft\foo{a}%
         \passexpanded\iffstring\foo{aaaa}
                      \breaklooptrue}
\foo
```

aaaa

`\newwhile`*<command><number><transformations><code>*

The `\dowhile` macro is not very powerful since you must generally change something somewhere to make it stops, and thus its expandability is somewhat perfunctory. That's why there is `\newwhile`. It creates an expandable *<command>* which takes *<number>* arguments (up to 9, as usual) and repeats *<code>* indefinetely. So, at first sight, it's bad news. But the point is *<code>* is supposed to launch the `\breakwhile` macro below sooner or later, i.e. stop the loop. Besides, on each iteration (barring the first), *<transformations>* are applied to the arguments, and this means: the first argument is replaced by the first transformation, the second argument by the second transformation, etc. So there must be as many transformations as there are arguments, transformations themselves being just code that can make reference to the arguments. If you don't want to transform an argument, just repeat it in the transformation.

```
\edef\foo{%
  The inconvenience of iff...:
  \dowhile{\straighteniff{ifnum}{4=5 }}
          {whatever}
  }
\meaning\foo
```

macro:->The inconvenience of iff...: {}

```
                        % Transformations.
\newwhile\largestsquare2{\numexpr(#1+1)}{#2}{%
  \reverse\straighteniff{ifnum}{\numexpr(#1*#1)<#2 }
                       {The largest number whose square
                        is smaller than #2 is
                        \breakwhile{\the\numexpr(#1-1).}}}
\largestsquare{1}{50}\par
\largestsquare{1}{200}
```

The largest number whose square is smaller than 50 is 7.
The largest number whose square is smaller than 200 is 14.

`\breakwhile`*<code>*

This breaks the current while loop and executes *<code>*, which can make reference to the arguments of the loop.

`\changewhile`*<new arguments>*

This replaces the default *<transformations>* defined with `\newwhile` and passes the *<new arguments>* for the next iteration. There must be as many arguments as required by the loop. The original *<transformations>* remain in force for the next iterations.

FOR STATEMENTS ON THE FLY

`\dofor<list><parameter text>{<definition>}<coda>`

This runs *<definition>* on each occurrence of *<parameter text>* in *<list>*. The *<parameter text>* is a real one, hence the braces around *<definition>*. The *<coda>* is executed if and only if the loop goes to its natural end, i.e. it is not terminated by one of the commands below. It must be present, even if you don't want one (in which case, leave it empty), and it can't make any reference to the arguments of the parameter text. A loop thus executed is absolutely not expandable. You can embed as many loops as you want (but don't forget to double the #).

```
\dofor{a,b,c,}#1,{[#1]}{}
```

[a][b][c]

`\dofornoempty`

This is the same as above, except *<definition>* is not executed when the *first* argument is empty.

```
\dofor{(a=13)(b=3)(c=54)(d=33)(e=22)}(#1=#2){%
  \straighteniff{ifnum}{#2>50 }
    {\breakfor{There's a number larger than 50: #1=#2.}}}
  {No number larger than 50.}
```

There's a number larger than 50: c=54.

`\breakfor<code>`

This breaks the current loop and executes *<code>*; the *<coda>* of the loop is not executed.

```
\dofornoempty{dd,e,,acb,3,ee4,,,}#1,{%
  \dofor{#1}##1{[##1]}{}...%
  }{}
```

[d][d]...[e]...[a][c][b]...[3]...[e][e][4]...

`\retrieverest<code>`

This also breaks the loop, but it retrieves the remaining arguments in the list and pass them as a braced argument to *<code>*.

`\pausefor<code>`

This interrupts the loop and executes *<code>*; the loop being interrupted means you're in the middle of the list, and you can process it. Such a pause must be terminated by a `\resumefor` if you don't want nasty internal code to surface.

`\resumefor\dofor`

This restarts the current loop. It is necessary to specify `\dofor`, because `\resumefor` is more general and is used to restart any kind of loop, especially those defined with `\newfor` (see next page).

The `\dofor` loop does not perform any kind of normalisation on the list. I.e. the list must be exactly designed to match the parameter text, including spaces and other unwelcome guests.

The `\dofor` macro is useful for straightforward loops used once or twice in a document. But for fully fledged total-control fully expandable hey-that's-too-cool loops, you should use the `\newfor` construction.

\newfor<*command*>{<*optional passed arguments*>}<*parameter text*>{<*definition*>}[<*optional coda*>]

This creates a recursive <*command*> that will consume all input with structure <*parameter text*>. Let's forget {<*optional arguments*>} for a while, since they're optional (albeit braced). Let's forget the optional coda as well. So it boils down to:

\newfor<*command*><*parameter text*>{<*definition*>}

so that basically \newfor works like \def. The <*parameter text*> is a real parameter text as with \def, just like {<*definition*>} is a real definition, hence the braces. The only difference is there must be at least one argument, because we need something to loop upon. I.e. <*parameter text*> is at least #1.

Now you can launch <*command*> on an argument which is made of as many occurrences of <*parameter text*> as you wish, and on each occurrence <*definition*> will be executed. So you've created a loop. And the good news is that this loop is fully expandable.

It is your job to make sure that what is fed to <*command*> has the correct argument structure.

If <*coda*> is specified, it is executed when the loop ends, if it ends naturally, i.e. by exhausting its input, and not by some of the loop-breaking commands on the next page. There can be no call to arguments of <*parameter text*> in the <*coda*>, e.g.

\newfor\foo#1{...}[...#1...]

is impossible. (You'll get raw inner code.) Such reference to arguments in the <*coda*> is possible only with passed arguments, as you'll see in two pages from here.

(Note that if there's no <*coda*>, any space will be gobbled after {<*definition*>}. This is so because I thought it was better to be able to write [<*code*>] after a space, e.g. a line end, than to stick it to {<*definition*>}, even though that brings this little inconvenience, which is probably harmless since \newfor is very unlikely to end up anywhere in horizontal mode, i.e. in a paragraph.)

Macro thus created can be freely embedded into one another.

\newfornoempty

The \newfornoempty is similar to \newfor, except <*definition*> is not executed in the case the *first* argument is empty.

```
\newfor\foo#1,{(#1)}
\foo{a,b,c,}
```

(a)(b)(c)

```
\newfornoempty\foo(#1,#2){[#1/#2]}
                          [Input exhausted.]
\edef\bar{\foo{(a,b)(c,d)(,e)(f,)}}
\meaning\bar
```

macro:->[a/b][c/d][f/]Input exhausted.

```
\newfor\values#1=#2,{%
  The value of #1 is #2.\par
  }
\def\setvalues#1{%
  \ifsuffix,{#1}{\values{#1}}
                {\values{#1,}}%
  }
\setvalues{A=12,B=45,}
\setvalues{C=34}
```

The value of A is 12.
The value of B is 45.
The value of C is 34.

```
\newstring, % \pdfliteral requires full expansion!
\def\drawline#1{
  0 0 m % Initializes the path
  \ifsuffix,{#1}{\drawlinefor{#1}}
                {\drawlinefor{#1,}}
  }                             % l = line
\newfor\drawlinefor#1 #2,{#1 #2 l }[S]% S = draw path
\pdfliteral{
  q  % kind of PDF \bgroup
  1 0 0 RG \drawline{20 10, 40 -15, 100 0,}
  0 1 0 RG \drawline{30 -15, 60 10, 130 0}
  Q} % kind of PDF \egroup
```

FOR STATEMENTS: INTERRUPTIONS

(The commands on this page are the same as those introduced with \dofor; they're explained more thoroughly here.)

`\breakfor`*<code>* Used inside a loop created with \newfor, this interrupts it, gobbles the remaining input, and executes *<code>*. If the loop had a *<coda>*, it is not executed. Any material between the \breakfor command and the end of the definition of the loop is gobbled. It is especially bad with conditionals, so you should use \afterfi, or better yet a \straightenif version.

`\retrieverest`*<code>* This does the same thing as \breakfor, i.e. breaks the current loop, but it passes the rest of the material initially passed to the loop as a braced argument to *<code>*. Arguments in that remaining material aren't extracted from their surrounding delimiters, if any.

`\pausefor` *<code>*
`\resumefor`*<loop command>* The \pausefor command stops the loop and executes *<code>*. That means that you're in the middle of the material being processed and you can act on it. It is useful if the material isn't totally regular. For instance, a typical BibTEX entry is a list of '*<field>*=*<value>*' pairs, with each pair terminated by a comma and the *<value>* either between braces or quotes. Thus, you can't have a simple

```
\newfor\bibfor#1=#2,{#1...#2}
```

to process the entry, because a *<value>* may be delimited by quotes and still contain a comma, and quotes mean nothing to TEX, so the comma will be mistaken for the delimiter. An oversimplified solution with \pausefor can be seen on the right. The loop actually works on the predictable part only (before the equal sign), is interrupted, the value is retrieved, and the loop is resumed. (Why one would want to process a BibTEX entry with TEX in the first place is a question I can personally answer.)

Once \pausefor is used, there must be somewhere down your code a \resumefor*<command>* statement, to launch the loop again, otherwise you'll end up stumbling on some nasty internal code. It is impossible to know (in a perfectly expandable way) the loop we're currently in, hence the *<command>* as a argument to \resumefor: it is the loop one wants to start again. Yes, it means you can also process the rest of the material with another loop, the consequences of which I leave it to you to ponder.

```
\newfor\foo#1{%
  \straighteniff{if}{\noexpand#1z}
    {\breakfor{There is a 'z'!}}
    #1... % This will be gobbled.
  }[There is no 'z'...]

\foo{abcdef}\par
\foo{abzdef}
```

a... b... c... d... e... f... There is no 'z'...
a... b... There is a 'z'!

```
\def\remainder#1{(And '#1' was still to come.)}
\newfor\foo#1=#2,{%
  \unless\ifnum#1=#2
    \afterfi{% Thrilling...
      \retrieverest{There is a false equation!
                    \remainder}}%
  \fi}
\foo{3=3,2=2,451=451,7=4,78=78,9=0,}
```

There is a false equation! (And '78=78,9=0,' was still to come.)

```
\newfor\bibfor#1={%
  \pausefor{\getvalue{#1}}}
\def\getvalue#1{%
  \trim{#1}:
  \ifnextnospace"{\getquotevalue}
                 {\getcommavalue}
  }
\def\getquotevalue"#1",{\showvalue{#1}}
\def\getcommavalue#1,{\showvalue{#1}}
\def\showvalue#1{%
  {\it\trim{#1}}.\par\resumefor\bibfor}

\bibfor{
  Author = {John Doe},
  Title  = "Me, myself and I",
  Year   = 1978,}
```

Author: John Doe.
Title: Me, myself and I.
Year: 1978.

FOR STATEMENTS: PASSED ARGUMENTS

Suppose you want to retrieve the largest number in a list a numbers. The first example on the right shows you how to do so. But this solution won't work if you need the loop to be expandable, because there's a number assignment.

That's why loops defined with `\newfor` can pass arguments from one iteration to the next. The number of those arguments are the {*<optional passed arguments>*} in the description of `\newfor` two pages ago. So, a typical fully-fledged use of `\newfor` is:

```
\newfor\myloop{2}#3=#4,{...#1...#2...#3...#4...}
                       [...#1...#2...]
```

which means that `\myloop` takes four arguments, two of which are actually passed arguments, the third and the four being in the recursive list that `\myloop` runs on. Besides, as you can see, passed arguments can appear in *<coda>*. Now a call to `\myloop` looks like:

```
\myloop{one}{two}{a=1,b=2,...}
```

where one and two are passed arguments. There can be up to 8 passed arguments (since there must be at least one argument to loop on), and if there are *n* of them, numbering of arguments in *<parameter text>* must start at *n*+1, as in the above example.

`\passarguments`*<arg1><arg2>...*

Passed arguments are automatically retrieved from one iteration to the next. However, if you can't change them, they aren't very interesting. Hence this command: it passes *<arg1>*, *<arg2>*, etc., to the next iteration, replacing the previous ones. There must be as many arguments to `\passarguments` as required by the loop, even if you don't want to pass new values for all (in which case, just pass the previous value). Beware: `\passarguments` ends the current iteration, just like `\breakfor`, and any remaining material in the definition of the loop is gobbled.

Thus, the second version of our `\findlargest` command works as follows: it takes one harmless passed argument, and loops on the following list. Obviously, 45 is larger than 0, so it is passed as the new first argument; then, 33 is not larger than 45, so nothing happen, and 45 is implicitly passed again as the first argument, and so on and so forth, until finally the *<coda>* prints the largest number in the list. And, as illustrated by the `\edef`, everything expands nicely.

```
\newcount\largest
\newfor\findlargest#1,{%
  \ifnum#1>\largest
    \largest=#1
  \fi}
  [The largest number is \the\largest.]

\findlargest{45,33,1,4844,12,655,}
```

The largest number is 4844.

```
\newfor\findlargest{1}#2,{%
  \straighteniff{ifnum}{#2>#1 }
                {\passarguments{#2}}%
  }
  [The largest number is #1.]

\edef\foo{\findlargest{0}{45,33,1,4844,12,655,}}
\meaning\foo
```

macro:->The largest number is 4844.

FOR STATEMENTS: EXAMPLES

Loops created with \newfor are somewhat tricky to get a hand on, so here are some examples. First of all, you might think that it would be nice to be able to define a loop whose argument structure is defined but not its replacement text, so that you can call it on similar lists but with different operations. For instance, a generic loop that works on all comma-separated lists. You can't do that exactly with \newfor, but you can easily use passed arguments to do something similar, e.g.:

```
\newfor\commalist{1}#2,{#1{#2}}
\commalist\tree{leaf,fruit,twig,}
\commalist\scale{b minor,f sharp,whatever lydian}
```

with \tree and \scale defined to process one argument: \commalist itself has no real definition, and you don't have to bother about passed arguments (although you can still use them).

The first example sorts a list of numbers separated by commas. The first loop, \sortnum, takes a passed argument which contains the numbers already sorted (so it is empty at the beginning) and it runs on the list to be sorted. The second loop, \subsortnum, takes two passed arguments: the first one is the number under investigation, the second one is the list of numbers smaller than the number under investigation (so it is empty too at the beginning), and it is updated each time we find such a number as the third, non-passed arguments to \subsortnum, which is an element of the list of already sorted numbers as preserved in \sortnum's first passed argument... got that?

Let's follow some iterations. The first call is:

```
            % incoming arguments
\sortnum{}5,12,-161,3,0,63,22,-45,
```

and it calls

```
\subsortnum{5}{}{}
```

so that \subsortnum terminates immediately: it has no input. So it calls its coda:

```
\passarguments{5,}
```

(where 5 is really the first argument following the empty

```
\newfor\sortnum{1}#2,{%
  \subsortnum{#2}{}{#1}%
  }[Sorted list: #1]
\newfor\subsortnum{2}#3,{%
  \straightenif{ifnum}{#1<#3 }
         {\retrieverest{\passtosortnum{#2#1,#3,}}}
         {\passarguments{#1}{#2#3,}}%
  }[\passarguments{#2#1,}]
\def\passtosortnum#1#2{\passarguments{#1#2}}

\sortnum{}{5,12,-161,3,0,63,22,-45,}
```

Sorted list: -161,-45,0,3,5,12,22,63,

second one). Since \subsortnum has terminated, this call to \passarguments is for \sortnum, hence the following iteration is:

```
                % incoming arguments
\sortnum{5,}12,-161,3,0,63,22,-45,
--> \subsortnum{12}{}5,
```

Ah, something new. 12 is larger than 5, so the conditional is false. So \subsortnum passes the following to itself:

```
\passarguments{12}{5,}
--> \subsortnum{12}{5,}{}
```

and once again it terminates, hence:

```
\passarguments{5,12,}   % incoming arguments
--> \sortnum{5,12,}-161,3,0,63,22,-45,
                          % incoming argument
--> \subsortnum{-161}{}5,12,
```

and obviously -161 is smaller than 5, so the rest of the list is retrieved with \retrieverest and passed as the second argument of \passtosortnum. Once again, since this terminates \subsortnum, \passarguments in \passtosortnum is for \sortnum:

```
\passtosortnum{-161,5,}{12,}
                           % incoming arguments
--> \sortnum{-161,5,12,}3,0,63,22,-45,
                          % incoming arguments
--> \subsortnum{3}{}-161,5,12,
--> \subsortnum{3}{-161,}5,12,
--> \passarguments{161,3,5,12,}
--> \sortnum{-161,3,5,12,}0,63,22,-45,
...
```

and so on and so forth.

Replace the test with any other one and you have a generic sorting function, as in the example on the right, which sorts entries alphabetically or chronologically. It is possible to make things both cleverer and simpler. (The Lua code compares two strings, and it could very well have handled the \year version.)

```
\newfor\sortbooks{2}#3(#4),{%
  \subsortbooks#1{#3(#4)}{}{#2}
  }[\bgroup\it#2\egroup]
\newfor\subsortbooks{3}#4(#5),{%
  #1#2{#4}{#5}{#3}
  }[\passarguments#1{#3#2,}]

\def\alpha#1(#2)#3#4#5{%
  \directlua{
    if "#1"<"#3" then
      tex.print("\noexpand\\firstoftwo")
    else
      tex.print("\noexpand\\secondoftwo")
    end}
    {\retrieverest{%
      \passtosortbooks\alpha{#5#1(#2),#3(#4),}}}
    {\passarguments\alpha{#1(#2)}{#5#3(#4),}}
  }
\def\year#1(#2)#3#4#5{
  \straightenif{ifnum}{#2<#4 }
      {\retrieverest{%
        \passtosortbooks\year{#5#1(#2),#3(#4),}}}
      {\passarguments\year{#1(#2)}{#5#3(#4),}}
  }
\def\passtosortbooks#1#2#3{\passarguments#1{#2#3}}

\def\books{
  Oblivion (2004),
  Infinite Jest (1996),
  Brief Interviews with Hideous Men (1999),
  Girl with Curious Hair (1989),
  The Broom of the System (1987),
  The Pale King (2011),
  }
David Foster Wallace's books in alphabetical order:\par
\passexpanded{\sortbooks\alpha{}}\books \par
David Foster Wallace's books ordered by date:\par
\passexpanded{\sortbooks\year{}}\books
```

David Foster Wallace's books in alphabetical order:
Brief Interviews with Hideous Men (1999), Girl with Curious Hair (1989), Infinite Jest (1996), Oblivion (2004), The Broom of the System (1987), The Pale King (2011),
David Foster Wallace's books ordered by date:
The Broom of the System (1987), Girl with Curious Hair (1989), Infinite Jest (1996), Brief Interviews with Hideous Men (1999), Oblivion (2004), The Pale King (2011),

FOR STATEMENTS: EXAMPLES

The next example is a palindrome detector: it returns true if the string it is fed is made of a string followed by itself reverse (which is not the exact definition of a palindrome, which is a string that is its own reverse, but we keep things simple).

The first loop, `\palincount`, simply counts the number of characters in the string; it also reaccumulates it as its second argument, something that could be avoided if there was a wrapper macro. Once it is finished, it passes the original string along with half the number of characters to `\palincheck`, which simply accumulates in reverse this number of characters, by decreasing it on each iteration. Once this number is exhausted, it compares what it has accumulated to what there remains to be processed, and if both strings match, the original string is a palindrome.

```
\newfor\palincount{2}#3{%
  \passarguments{\numexpr(#1+1)}{#2#3}%
  }[\palincheck{\numexpr(#1/2)}{}{#2}]
\newfor\palincheck{2}#3{%
  \reverse\straightenif{ifnum}{\numexpr(#1-1)>0 }
                {\retrieverest{\compare{#3#2}}}
                {\passarguments{\numexpr(#1-1)}{#3#2}}%
  }
\def\compare#1#2{%
  \ifstring{#1}{#2}{TRUE}{FALSE}%
  }

\edef\foo{\palincount{0}{}{abcdeffedcba}}
\edef\bar{\palincount{0}{}{abcdff}}
\meaning\foo, \meaning\bar
```

macro:->TRUE, macro:->FALSE