Creating macros in OpTEX

Petr Olšák

Introduction

OpTEX [1] is an extended plain TEX. We can create macros as in plain TEX. In particular, this means that we use TEX primitives like `\def`, `\edef`, `\ifx`, `\expandafter`, `\csname`, `\hbox`, `\vbox`, `\hrule`, and so on. Likewise, we use basic plain TEX macros like `\newcount`, `\llap` and many others. I wrote a summary of these TEX and plain TEX tools in [2].

OpTEX keeps the plain TEX philosophy: it does not create any new syntactic, semantic, or thought layers over TEX, so the commands mentioned above are principal ones, basic for creating macros. For example, OpTEX doesn't try to provide anything similar to `\newcommand`, nor anything similar to `expl3`. The main message is: if you know TEX, you can make your macros.

On the other hand, OpTEX provides many elementary macros which can make macro programming easier. And there are a few conceptual recommendations especially to separate different namespaces when your macros will be used for public purposes. This article summarizes these tools and principles. More detailed information can be found in the OpTEX manual [3].

Naming conventions and namespaces

When you are creating macros for your use then you can use arbitrary alphabetical names for newly declared control sequences. Moreover, you can redefine existing names, if you decide that it is useful and you never are using them in their original meaning. For example, you can define `\def\box{...}` without any problem, as long as you use it only with your declared meaning. It doesn't matter that `\box` is a TEX primitive in its original meaning. OpTEX internally uses copies of all primitive names and internal macro names, `\_box` in this case.

In other words, when OpTEX starts, all internal sequences are duplicated (both `\box` and `\_box` are present, with the same meaning) and OpTEX uses only the name `\_box` in its internal macros. A user can redefine `\box` if he or she finds it useful, or doesn't know that the name is already used. There is only one requirement: if you re-declare a control sequence then it cannot be used in its original meaning in your document. For example, `\def\def{...}` is possible but then you cannot use `\def` as a primitive command in your next text.

The alphabetical control sequences like `\foo`, `\SomethingOther`, `\hbox` are considered in "public namespace" from the OpTEX point of view. On the other hand, alphabetical control sequences beginning with "`_`" (like `\_foo`) are reserved in special namespaces. The character "`_`" has category code 11 (letter) in OpTEX.* This means that you have full access to the internal control sequences like `\_foo` without any category dancing. You don't need to say something like `\makeatletter ... \makeatother`. You can use these internal sequences in "read-only" mode without any restrictions. You can redefine them too, but if you decide to do that then you hopefully know what you are doing, and what internal process in OpTEX will be changed.

As mentioned above, "OpTEX's private namespace" includes names beginning with "`_`" followed by normal letters. There are copies of all TEX primitives and internal OpTEX macros here. An internal macro of a macro package uses its "package's private namespace" `\_pkg_foo`, where "`pkg`" is a shortcut of the package. A package writer uses the `\_namespace` declaration for dealing with such control sequences more comfortably; see below, the section "writing public macro packages".

The single-letter control sequences (like `\$`, `\/`, `\,`) are declared similarly to plain TEX** and are not used in internal OpTEX macros. Users can re-declare them freely without affecting the behavior of OpTEX.

Basic macros for macro programmers

OpTEX provides a few basic macros:

- `\sdef{`⟨*cs-name*⟩`}` defines a control sequence whose name is given by the ⟨*cs-name*⟩ string. Thus, `\sdef {T\the\mycount}#1:#2{...}` is (roughly speaking) an equivalent to `\def\T42 #1:#2{...}` if `\mycount=42`.
- `\sxdef{`⟨*cs-name*⟩`}` is similar to `\sdef`, but the `\xdef` primitive is used behind the scenes instead of `\def`.
- `\slet{`⟨*cs-name1*⟩`}{`⟨*cs-name2*⟩`}` is (roughly speaking) equivalent to `\let` ⟨*cs-name1*⟩`=` ⟨*cs-name2*⟩.

* There is a little trick to enable use of this character with its normal plain TEX meaning in math mode without changing this category. But it works.

** Not all single letter control sequences from plain TEX are available. Control sequences for accents like `\"`, `\'` are undefined by default because we suppose that accented letters are directly written in Unicode (the current year is 2023). But a conservative user can enable them with the `\oldaccents` declaration.

doi.org/10.47397/tb/44-1/tb136olsak-optexmac

- `\adef` ⟨*character*⟩ sets the ⟨*character*⟩ to be active and defines it like `\def`⟨*character*⟩. For example, `\adef *{...}` or `\adef @#1#2{...}`.
- `\optdef\macro [`⟨*default*⟩`]`⟨*params*⟩ is similar to `\def\macro` ⟨*params*⟩ but the `\macro` can be used with an optional argument given like `\macro[`⟨*text*⟩`]` before scanning other parameters. If the optional syntax is used then the token register `\opt` includes ⟨*text*⟩. Otherwise, it includes ⟨*default*⟩.
- `\eoldef\macro #1{...}` defines `\macro` with a single parameter delimited by the end of the current line. This parameter is `#1`, and it can be used in the macro body. Such a `\macro` can be used only when lines of text are read, not inside other macros.
- `\cs{`⟨*text*⟩`}` is a shortcut for the commonly-used `\csname` ⟨*text*⟩`\endcsname`.
- `\trycs{`⟨*text*⟩`}{`⟨*else*⟩`}` does `\cs{`⟨*text*⟩`}` only if `\cs` is defined, otherwise the ⟨*else*⟩ part is processed.

There is a useful shortcut of the `\expandafter` primitive: `\ea`. Of course, it is safer to use `\_ea` because control sequences with short names tend to be re-declared later by a user.

Additional simple macros include:

- `\ignoreit{`⟨*text*⟩`}` does nothing.
- `\useit{`⟨*text*⟩`}` does ⟨*text*⟩.
- `\ignoresecond{`⟨*A*⟩`}{`⟨*B*⟩`}` does ⟨*A*⟩.
- `\usesecond{`⟨*A*⟩`}{`⟨*B*⟩`}` does ⟨*B*⟩.

You can add a given text to a parameterless macro:

- `\addto\macro{`⟨*text*⟩`}` appends ⟨*text*⟩ to the `\macro` body.
- `\aheadto\macro{`⟨*text*⟩`}` prepends ⟨*text*⟩ to the `\macro` body.

You can globally increase a counter by one by `\incr`⟨*counter*⟩ and decrease it by `\decr`⟨*counter*⟩. `\opwarning{`⟨*message*⟩`}` prints a message to the terminal and log file.

Branching macro processing

Of course, you can use all of TeX's `\if*` primitives. OpTeX also provides the following `\is*` conditionals with the general syntax being one of:

`\isfoo...\iftrue` ⟨*true-text*⟩`\else` ⟨*false-text*⟩`\fi`
or
`\isfoo...\iffalse` ⟨*false-text*⟩`\else` ⟨*true-text*⟩`\fi`

The macro `\isfoo` calculates the condition and gobbles the `\iftrue` or `\iffalse`. You have to use this syntax because the `\isfoo` block can be skipped by another outer `\if` condition and the pairs `\if`...`\fi` must match.

The `\isfoo` macros are:

- `\isempty{`⟨*text*⟩`}\iftrue` is true if ⟨*text*⟩ is empty.
- `\isequal{`⟨*text-A*⟩`}{`⟨*text-B*⟩`}\iftrue` is true if the string ⟨*text-A*⟩ is equal to ⟨*text-B*⟩. These parameters are treated as strings. The category code of the characters has no effect.
- `\ismacro\macro{`⟨*text*⟩`}\iftrue` is true if the body of the parameterless `\macro` is equal to given ⟨*text*⟩.
- `\isdefined{`⟨*cs-name*⟩`}\iftrue` is true if the ⟨*cs-name*⟩ is defined.
- `\isinlist\list{`⟨*text*⟩`}\iftrue` is true if the ⟨*text*⟩ is included in the `\list` macro body.
- `\isfile{`⟨*file-name*⟩`}\iftrue` is true if the file named ⟨*file-name*⟩ exists and is accessible by TeX for reading.
- `\isfont{`⟨*font-name*⟩`}\iftrue` is true if the font named ⟨*font-name*⟩ exists. You can use `[`⟨*font-file*⟩`]` instead of ⟨*font-name*⟩ too.

All these `\is`-macros are fully expandable. The following `\is`-macro has different syntax than the macros mentioned above, but it is also expandable:

- `\isnextchar` ⟨*char*⟩`{`⟨*true-text*⟩`}{`⟨*false-text*⟩`}`. If the next character is equal to ⟨*char*⟩ then ⟨*true-text*⟩ is processed else ⟨*false-text*⟩ is processed.

OpTeX provides the `\afterfi{`⟨*text*⟩`}` macro which can be used inside `\if`...`\else`...`\fi`. The macro closes the `\if`...`\fi` block and runs ⟨*text*⟩ after it is closed. For example

```
\ifx\a\b ... \afterfi{do something}%
   \else ... \afterfi{do something else}%
\fi
```

This is almost the same as `\_ea` ⟨*token*⟩`\else` or `\_ea` ⟨*token*⟩`\fi` but the `\afterfi` parameter can include more than a single token.

A nested `\if`...`\fi` block can be inside the `\afterfi` parameter and `\afterfi` macros can be here too. It means that nested `\afterfi` macros work as expected. You don't need to escape from a nested `\if`...`\fi` block by a larger number of `\expandafter`s.

We must recall that usage of primitive conditionals with `\if`...`\fi` blocks hides one potential problem: if you are designing a macro that reads a TeX macro code token by token then your `#1` might be `\if` or `\else` or something similar. Usage of such `#1` inside your `\if`...`\fi` block in your macro causes TeX to give an error. What can you do in such a case? Here's one example, when looking for a specific token (~, here):

```
\ifx ~#1\ea\ignoresecond\else \ea\ignorefirst\fi
   {⟨true text with #1, we know that #1 is ~⟩}
   {⟨else text with #1⟩}%
```

Branching with more structural macros

OpTEX provides macros `\caseof` and `\xcaseof` to switch among more alternatives. Usage of `\caseof`:

```
\caseof ⟨token⟩
   ⟨token A⟩ {⟨text A⟩}
   ⟨token B⟩ {⟨text B⟩}
   ⟨token C⟩ {⟨text C⟩}
   ...
   \_finc   {⟨else text⟩}%
```

If ⟨*token*⟩ is ⟨*token A*⟩ then only ⟨*text A*⟩ is processed, if ⟨*token*⟩ is ⟨*token B*⟩ then only ⟨*text B*⟩ is processed, etc. If ⟨*token*⟩ differs from all declared tokens, then ⟨*else text*⟩ is processed.

The `\xcaseof` macro is similar, but you can specify an arbitrary primitive `\if`-test instead of a token comparison only. The fragment of the code above with one condition more can be written as

```
\let\next=⟨token⟩
\xcaseof
   {\ifx\next A}        {⟨text A⟩}
   {\ifx\next B}        {⟨text B⟩}
   {\ifx\next C}        {⟨text C⟩}
   {\ifnum\mynum=12 } {⟨text 12⟩}
   ...
   \_finc {⟨else text⟩}%
```

A `\caseof` block is skippable by an outer `\if`...`\fi` block but `\xcaseof` is not.

If there is more than one "true" result of the conditions given by `\xcaseof`, then the first condition wins and the others are skipped.

The `\_finc` separator followed by `{`⟨*else text*⟩`}` is obligatory. Of course, you can declare empty ⟨*else text*⟩. The separator must be written as `\_finc` separator, not `\finc`. The reason is that the same syntax is given for `\_caseof` and `\_xcaseof` macros.

The spaces between `\caseof` or `\xcaseof` parameters are ignored but not the last space after the `{`⟨*else text*⟩`}`. Note the percent character in the examples.

`\caseof` and `\xcaseof` are fully expandable macros.

Loops

You can use the classical plain TEX `\loop` macro. The one difference from plain's `\loop` is that OpTEX allows you to declare `\if`...`\else`...`\repeat`. But nothing more. There are still limitations here: `\loop` is not expandable and `\loop` inside `\loop` is possible only if the inner `\loop` is in a group.

OpTEX provides two additional looping macros, `\foreach` and `\fornum`. They are fully expandable and can be arbitrarily nested without declaring a group. The body of these macros is processed without opening and closing a group. The syntax of the `\foreach` macro is one of:

```
\foreach ⟨text⟩\do {⟨body⟩}
 or
\foreach ⟨text⟩\do ⟨parameter-spec⟩{⟨body⟩}
```

The first variant runs ⟨*body*⟩ for each token* from the ⟨*text*⟩. The current token (current parameter) can be processed inside the ⟨*body*⟩ as `#1`. If the `\foreach` block is included inside another macro then you have to use `##1`; if it is inside a macro in a macro, or inside another `\foreach` or `\fornum` body, then use `####1`, etc.

The second variant with ⟨*parameter-spec*⟩ enables scanning of the given ⟨*text*⟩ with an arbitrary parameter specification like with `\def`. You can declare separators for these parameters. For example, suppose we are creating a macro `\macro` which gets a parameter as a list of pairs in parentheses:

```
\macro{(a,b); (c,d); (1,42)}
```

and we want to read this and print these pairs in reverse order and with a different format: `b/a d/c 42/1`. We can do this by:

```
\def\macro#1{%
   \foreach #1\do ##1(##2,##3){##3/##2 }%
}
```

The unused `##1` is there because we want to ignore an optional "`; `" before the opening `(`.

This `\macro` is expandable, so you can use it inside the parameter of the `\message` primitive, for example.

The `\fornum` and `\fornumstep` macros have the following syntax:

```
\fornum ⟨from⟩..⟨to⟩ \do {⟨body⟩}
 or
\fornumstep ⟨step⟩: ⟨from⟩..⟨to⟩ \do {⟨body⟩}
```

The ⟨*body*⟩ is repeated for numbers starting at ⟨*from*⟩ and ending at ⟨*to*⟩. The `\fornum` increments the number by one. The second case uses the given ⟨*step*⟩. The parameters ⟨*from*⟩, ⟨*to*⟩, ⟨*step*⟩ can be

* Not always a single token: if the ⟨*text*⟩ includes `{...}` then all tokens inside these braces are taken at once, similar to the scanning of a macro parameter.

any arbitrary expression accepted by the `\numexpr` primitive. The current number is accessible in ⟨*body*⟩ as `#1` (or `##1` inside macros, etc.).

You may notice that there is a name conflict: the same control sequence `\foreach` is used by the Ti*k*Z package with different syntax and different features. OpTEX enables loading Ti*k*Z by `\load[tikz]`. If this is done then the `\foreach` from Ti*k*Z is available only within the `\tikzpicture`...`\endtikzpicture` environment. Outside this environment, the `\foreach` from OpTEX is active. Moreover, your macro code can use the private `\_foreach` from OpTEX if you want to be sure what you are using. With `\_foreach`, you have to use the `\_do` separator, instead of `\do`.

Why is there such a naming conflict? My macros are several decades old; older than Ti*k*Z. I don't want to rename this control sequence only due to Ti*k*Z (especially when I personally hardly use Ti*k*Z).

Key–value syntax for parameters

Calling `\readkv{`⟨*list*⟩`}` or `\readkv\list` reads a given ⟨*list*⟩ or a `\list` macro in ⟨*key*⟩=⟨*value*⟩ format. These pairs are comma-separated, and the =⟨*value*⟩ may be missing. Once the ⟨*list*⟩ is read, you can access the ⟨*value*⟩ by expandable macro `\kv{`⟨*key*⟩`}`. If you only need to know whether the ⟨*key*⟩ was used then `\iskv{`⟨*key*⟩`}\iftrue` returns the answer.

You can also declare ⟨*code*⟩ to be processed whenever a particular ⟨*key*⟩ is encountered during `\readkv`. This is done with `\kvx{`⟨*key*⟩`}{`⟨*code*⟩`}`. The ⟨*code*⟩ can access the scanned ⟨*value*⟩ as `#1`. Specifying `\nokvx{`⟨*other code*⟩`}` declares common ⟨*other code*⟩ to process for all ⟨*keys*⟩ not declared by `\kvx`. The ⟨*other code*⟩ can use `#1` to access the ⟨*key*⟩ and `#2` to access the ⟨*value*⟩.

The ⟨*key*⟩=⟨*value*⟩ data are stored in and read from a dictionary with the name `\kvdict`, which is a token register. It is empty by default, i. e. the default dictionary has an empty name. You can manage more dictionaries by changing it.

The following example is borrowed from the OpTEX documentation. We define a macro `\myframe` which can scan optional parameters in `[...]` key–value format and sets colors and dimensions by these parameters. When we use, for example

```
\myframe [rule-width=2pt, frame-color=\Blue]
         {text}
```

then a frame around the given `text` with rule width 2pt in blue color is created. The macro can be defined like this:

```
\def\myframedefaults{% defaults:
   frame-color=\Black, % color of rules
   text-color=\Black,  % color of the text
   rule-width=0.4pt,   % width of rules
   margins=2pt, % space between text and rules
}
\optdef\myframe []#1{%
   \bgroup
   \readkv\myframedefaults \readkv{\the\opt}%
   \rulewidth=\kv{rule-width}%
   \hhkern=\kv{margins}%
   \vvkern=\kv{margins}\relax
   \kv{frame-color}%
   \frame{\kv{text-color}\strut #1}%
   \egroup
}
```

The `\myframe` macro from this example runs the `\frame` macro provided by OpTEX. Its parameters `\rulewidth`, `\hhkern` and `\vvkern` are set from values given in key–value format when `\myframe` is used. The `\myframedefaults` macro clearly specifies the default values and any user-given values are read from the optional argument from the `\opt` tokens register. The `\readkv` macro is used twice: first the default values are read and second, the user-specified values are read. The last assignment wins.

Expressions

In addition to the well-known `\numexpr` primitive from ε-TEX, OpTEX provides the expandable macro `\expr{`⟨*expression*⟩`}`, which calls the Lua interpreter (OpTEX always runs under LuaTEX) and does arithmetic with decimal numbers. The number of decimal digits of the result is 3 by default; this can be overridden with the optional argument, as in `\expr[`⟨*digits*⟩`]{`⟨*expression*⟩`}`. Examples:

```
\expr{2*(4-1.3)}      % 5.400
\expr{math.sqrt(1/3)} % 0.577
\expr[14]{math.pi}    % 3.14159265358979
```

The OpTEX macro `\bp{`⟨*dimen*⟩`}` provides an expandable conversion of ⟨*dimen*⟩ to the decimal number which expresses the given value in `bp` units. The ⟨*dimen*⟩ can be an arbitrary expression accepted by the `\dimexpr` primitive. The result is a decimal number without a unit. It can be used in, for example, arguments to the `\pdfliteral` literal where we are using such numbers without units in low-level PDF commands. For example, `\bp{\parindent}` returns `19.925` in this document, because `\parindent` is set to 20pt here.

You can use the `\bp{`⟨*dimen*⟩`}` as operands in the `\expr{`⟨*expression*⟩`}`. This is very useful when we are programming graphics using `\pdfliteral`.

More programming tools

The list of macros provided for macro programmers cannot be complete in this short article. There are many macros specialized for particular problems like math macros, color macros, font macros, reference macros, citation–bib macros, etc. See the OpTEX documentation [3] for more information. There are plenty of macro programming tips on the OpTEX tricks page [4] too. Here, I will demonstrate only two more cases of useful macros.

First case: `\replstring\buff{`⟨*from*⟩`}{`⟨*to*⟩`}` replaces all occurrences of ⟨*from*⟩ text by ⟨*to*⟩ text in the "buffer" macro `\buff`. For example:

```
\def\buff{A text is here.}
\replstring\buff{ }{{ }}
\ea\foreach\buff \do{[#1]}
It returns: [A][ ][t][e][x][t][ ]%
            [i][s][ ][h][e][r][e][.]
```

We have used `\replstring` in this example to "protect" spaces. Each space is replaced by `{ }`. So, the next macro `\foreach` (which reads token by token via an internal macro taking an undelimited parameter `#1`) can read spaces too.

Second case: We can set the current typesetting position anywhere by `\setpos[`⟨*label*⟩`]` and then read this position elsewhere with the (expandable) commands `\posx[`⟨*label*⟩`]`, `\posy[`⟨*label*⟩`]` and `\pospg[`⟨*label*⟩`]`. The first two commands return the x, y coordinates of the absolute position of the `\setpos` point on the page (measured from the left-bottom corner). The values are given in the format ⟨*number*⟩`sp`. You can convert to `bp` units (for example) with `\bp{\posx[`⟨*label*⟩`]}` or read into a variable with `\mydimen=\posx[`⟨*label*⟩`]\relax`. The last one (`\pospg`) returns the global page number of the document where the `\setpos` point was set. The data is available after a second run of TEX because an external `.ref` file is used for this purpose.

Writing public macro packages

When you are writing macros for your usage, there are no rules for naming the control sequences. You can write any macro code and test it. If you plan to release such a macro code as a public package, however, then I recommend the following naming conventions described here. You can look at the code of the `math.opm` package [5] for inspirations and examples of how to create packages for OpTEX. This package deals with options, math macros, and there is a special section about writing public packages too.

First of all, you may set a package shortcut. I'll use the shortcut `pkg` in the following examples. If you select a shortcut used by another package, then users are unable to load both these packages at one time: OpTEX reports an error. So, it is a good idea to see what public packages for OpTEX are available and thus choose a shortcut that isn't already being used.

First, two code lines (after optional comments) in the package file (which should be named `pkg.opm` for our example) should be

```
\_def\_pkg_version {0.07, 2023-01-14}
\_codedecl \supermacro {Title <\_pkg_version>}
```

The first argument of the `\_codedecl` macro (in this example, `\supermacro`) is a macro name that `\_codedecl` checks for being already defined; if it is, `\endinput` is executed, so that the package is not read twice. The idea is that `\supermacro` is a macro never used before and will be defined in this package. The second argument of the `\_codedecl` macro is printed to the log file. The `Title` should be a short title for the package.

The macro code that follows has to be surrounded by

```
\_namespace{⟨package-shortcut⟩}
...
\_endnamespace
```

I. e. `\_namespace{pkg}`...`\_endnamespace` in our example. Also, the `\_endcode` macro has to be called just after `\_endnamespace`. It is similar to `\endinput`, but has more features (described below).

Suppose that you have tested your macros with names in the public namespace. Now, rename all used control sequences by the following rules:

- If it is a TEX primitive or an OpTEX macro, add the "`_`" prefix: use `\_foo` instead of `\foo`.
- If it is your macro, defined and used in the package, add the "`.`" prefix.

Each `\.foo` is transformed to `\_pkg_foo` automatically inside the `\_namespace`...`\_endnamespace` scope. A macro programmer is thus not forced to write and read his package shortcut again and again for essentially all internal control sequences in the macro code.

If you decide that a macro is intended for users in the public namespace, export it from the package namespace to the public namespace using:
`\_nspublic`⟨*list of control sequences*⟩`;`
In our example, we could do:

```
\_def \.supermacro #1#2#3{...}
\_nspublic \supermacro ;
```

The `\_pkg_supermacro` and `\supermacro` control sequences are now defined, with the same meaning.

The `\_nspublic` command checks if the given macro is defined already in the public namespace. If so, then it is redefined, but a warning about it is shown on the terminal.

Maybe there is no reason to declare both the internal copy of a control sequence `\.foo` and the public copy `\foo`. You can declare `\foo` directly, as in `\_mathchardef\foo`, `\_newcount\foo`, `\_def\foo`, etc. But it is highly recommended to prefix such a declaration by `\_newpublic`. For example:

```
\_newpublic \_newcount \foo
\_newpublic \_mathchardef \bar = "123456
```

This prefix does the same check as `\_nspublic`: if a declared control sequence is already defined, it is redefined but with a warning printed.

You can add documentation text to individual macros in a `\_doc`...`\_cod` block. These parts are skipped when your macros are read. For example:

```
   \_doc
   The \`\supermacro` reads parameters and does
   a supertrick A and then does a supertrick B.
   \_cod

\_def \.supermacro #1#3#3{...A...B.}
\_nspublic \supermacro ;
```

It is recommended to append more extensive documentation of the package after the `\_endcode` command. This text is not read, because `\_endcode` executes `\endinput`. This way, you have code and documentation together in a single file, making it much more convenient to manage the package.

You can append a special block `\_doc`...`\_cod` to the documentation after `\_endcode`, to include commands used by the `\docgen` command from OpTEX. Typical usage of this final `\_doc`...`\_cod` scope could be:

```
\_doc
   \load [doc] % provides \printdoc, etc.

   \tit Package which enables super-\TeX/ing
   \hfill Version: \_pkg_version \par
   \centerline{\it Au. Thor\/\fnotemark1, 2023}
   \fnotetext{\url{https://au.thor.or}}

   \notoc\nonum\sec Table of contents
   \maketoc

   \printdoctail pkg.opm % prints the doc.
                 % written after \_endcode
   \sec Implementation
   \printdoc     pkg.opm % prints doc. of code
                 % before \_endcode

   \nonum\sec Index
   \begmulti 3
      \tt \makeindex % prints index, 3 columns
   \endmulti
   \bye
\_cod
```

The macros provided by `\load[doc]` are described in the OpTEX documentation [3], section 2.40.

Now, you have everything you need in a single file: the code itself, technical short documentation, detailed documentation, and commands to generate a whole document including title, table of contents, index, etc. The macro code is ready to be used directly without docstrip pre-processing.

A user can load your package with `\load[pkg]` or can generate a complete documentation by the command line:

```
optex -jobname pkg \\docgen pkg
```

You can try to use this command for the real existing package:

```
optex -jobname math \\docgen math
```

Run this command three times because TEX needs to generate the correct table of contents and the index.

If you write a package for OpTEX, please let me know about it. I'll add a notice about it into [3], section 1.7.3.

References

1. OpTEX. `petr.olsak.net/optex/`
2. P. Olšák: TEX in a Nutshell. 2020, 30 pp. `https://petr.olsak.net/ftp/olsak/optex/tex-nutshell.pdf`
3. OpTEX manual. `https://petr.olsak.net/ftp/olsak/optex/optex-doc.pdf`
4. OpTEX tricks. `https://petr.olsak.net/optex/optex-tricks.html`
5. OpTEX macros for doing math more comfortably. `https://petr.olsak.net/ftp/olsak/optex/math-doc.pdf`

◇ Petr Olšák
Czech Technical University
in Prague
`https://petr.olsak.net`