The template package*

DPC, FMi

2009/11/01

Contents

*This file has version number 1670, last revised 2009/11/01.

1 Introduction

A *Template* is a named 'function' that has a fixed number of *mandatory arguments* and an additional set of *keys* or 'named attributes' that are set in a 'key value list'. That is, a comma separated set of assignments of the form:
⟨*key*$_1$⟩ `=` ⟨*value*$_1$⟩ `,` ⟨*key*$_2$⟩ `=` ⟨*value*$_2$⟩ ...

More specific instances of the template may be declared by specifying settings of the parameters. The key value list is parsed at the time the instance is declared, and an 'internal' set of parameter assignments is passed to the template code. It is normally not parsed at run-time, though it is possible to enforce this behaviour.

Templates have a *type* and a *name*. Templates of the same type have the same argument and parameter structure. That is, templates of the same type are expected to be exchangeable semantically. (However, except for checking that templates of the same type always have the same number of arguments this is not enforced by the code.)

A template type is declared via the `\DeclareTemplateType` declaration which takes two arguments: the name for the type and the number of arguments a template of this type requires.

Templates are declared via the `\DeclareTemplate` command which takes five arguments

- The *type* of the template (no \).
- The *name* of the template (no \).
- The number of arguments for the template (same as on type declaration).
- A list declaring the keys accepted by the template, with information about the action to take when the key is specified with a value.
- The code for the template. This may be arbitrary TEX code. At some point it should execute `\DoParameterAssignments` to run the parameter assigments.

  The mandatory arguments are accessed via `#1`, `#2`, ...

Each element of the key specification list is of the form:

⟨*key name*⟩ `=` ⟨*key type*⟩ ⟨*optional default*⟩ ⟨*internal code*⟩

The ⟨*key types*⟩ are essentially specified by giving a symbolic representation of the assignment function to be used by TEX.

Currently the possibilities are

key type	letter	internal code	argument form
Command	f*n*	command name	command definition
name	n	command name	command definition
length	l	length register	calc length syntax
fake length	L	command name	calc length syntax
count	c	count register	calc count syntax
fake count	C	command name	calc count syntax
boolean	b	name of `\newif` switch	true or false
switch	s	`{`⟨*true code*⟩`}{`⟨*false code*⟩`}`	true or false
instance	`i{`⟨*type*⟩`}`	command name	instance of this *type*
direct	x	Internal code	any
general	g	general code	any

In addition, any of these types may be prefixed by `+` to denote a global assignment (described below). `f` takes a digit from 0–9 to denote the number of arguments. `n` is in fact the same as `f0`. When an instance is declared The value assigned to the key should be the definition of the command, using `#1...#9` to denote the specified arguments.

`c` takes an internal form a count register, not a LATEX counter name.

For `f`, `n`, `l`, and `c`, the assignment is done twice, once at the time an instance is *declared*. (This may involve using `calc` expresions. Then the 'primitive assignment' of the value (not using calc) is copied to the internal parameter list, to be executed when an instance is run. Sometimes you need the expression to be evaluated at the time an instance is run rather than the time it is declared. For example it may be an expression involving some values that are not fixed throughout a document. In this case the instance declaration may give a value in the form `\DelayedEvaluation{`⟨*calc expression*⟩`}`. In this case the value is not evaluated when the instance is declared, and instead the entire expression is copied to the 'internal parameter list' and is evaluated whenever the instance is used.

`L` and `C` take the same value types as `l` and `c` but the internal assignments are to macros not registers.

Keys declared with `b` and `s` each take values either *true* or *false*. if the key zzz is declared with `b` then specifying `zzz=true` will essentially pass `\zzztrue` to the internal parameter list (although in fact `\zzztrue` need not be defined) . If instead zzz had been declared via `s`, then `zzz=true` would pass the tokens of the `{`⟨*true code*⟩`}` to the internal parameter list.

If a key is specified as `x`, then when used the *internal code* will be copied to the internal parameter lists. This code may use `#1` to denote the value supplied to the key in the instance declaration. Note that this code is *only* copied at the time the instance is declared. It is not executed at this time. It is executed when the instance is executed.[1]

[1] Despite the question of whether or not x and g are still necessary these days, they have the wrong 'names' since x is the one that is not executed during delcaration while g is.

If a key is declared with `g` then the code is run at the time the instance is declared. By default *nothing* is passed to the internal parameter list. This code may use `#1` to denote the value that will be supplied when an instance is declared. Any code that should be run when an instance is executed should be explicitly passed to the internal parameter list using `\toks_add_right:Nn\l_TP_KV_assignments_toks{...}`

A key declared with `i{`⟨*type*⟩`}` takes as value the name of a declared instance of that type. The command token associated with the key will store a command essentially equivalent to a call to `\UseInstance{`⟨*type*⟩`}{`⟨*name*⟩`}`, but in a slightly optimised internal form. As an exception to this rule the replacement code may be of the form `\UseTemplate` followed by the key settings for the template but without the mandatory arguments. In this case the 'inner' instance declaration is 'pre compiled' and the token assigned to the store the value assigned to this key will execute an instance of the template directly, it will not re-parse the keyword settings each time the instance is used.

2 Commands

2.1 Template declaration commands

`\DeclareTemplateType {`⟨*type*⟩`}{`⟨*num*⟩`}`

Declare a template type.

`\DeclareTemplate {`⟨*type*⟩`}{`⟨*tname*⟩`}{`⟨*num*⟩`}{`⟨*keyspec*⟩`}{`⟨*code*⟩`}`

Declare a template ⟨*tname*⟩ of type ⟨*type*⟩ with the set of keys as defined by ⟨*keyspec*⟩. From this template instances can be declared using `\DeclareInstance` At runtime such instances will run ⟨*code*⟩ and expect ⟨*num*⟩ mandatory arguments (same number for all templates of one type).[2]

`\DeclareRestrictedTemplate {`⟨*type*⟩`}{`⟨*new-tname*⟩`}{`⟨*old-tname*⟩`}{`⟨*keyvals*⟩`}`

Declare as new template ⟨*new-tname*⟩ for type ⟨*type*⟩ by taking template ⟨*old-tname*⟩ as the basis and setting one or more of its keys to specific values.

`\DoParameterAssignments`

The list of key value assignments made (and saved) during template declaration is evaluated at this point in the template code.

[2]The ⟨*num*⟩ argument is redundent as it can be deduced from the type. However, for practical reasons it seems better to keep that information with each individual template declaration.

2.2 Instance declaration commands

`\DeclareInstance {`⟨*type*⟩`}{`⟨*iname*⟩`}{`⟨*tname*⟩`}{`⟨*keyvals*⟩`}`

Declare an instance of type ⟨*type*⟩ named ⟨*iname*⟩ build from using template ⟨*tname*⟩ with key settings as given by ⟨*keyvals*⟩.

`\DeclareCollectionInstance {`⟨*collection*⟩`}{`⟨*type*⟩`}{`⟨*iname*⟩`}{`⟨*tname*⟩`}{`⟨*keyvals*⟩`}`

Same as `\DeclareInstance` except that this instance is only active when for the type ⟨*type*⟩ the collection ⟨*collection*⟩ was selected via `\UseCollection`. E.g., within the frontmatter one could make all headings behave differently by defining collection instances for template type 'head'.

2.3 Key value commands

`\DelayEvaluation{`⟨*code*⟩`}`

Used in the value spec for an instance to declare that the value ⟨*code*⟩ should not be evaluated at declaration time but at run-time. Can also be used in the defaults for keys (given in square brackets) in the declaration of templates.

`\MultiSelection` ⟨*counter*⟩ `{`⟨*cases*⟩`} {`⟨*else*⟩`}`

Used in the value spec for an instance key to declare that the value of this key depends on the current setting of ⟨*counter*⟩ at run-time. The ⟨*cases*⟩ argument is a comma-separated list of "values", the ⟨*else*⟩ argument a single "value". If at run-time ⟨*counter*⟩ has the value i then the i-th element of the ⟨*cases*⟩ list is selected. If that does not exist the ⟨*else*⟩ case is returned.

2.4 Processing commands

`\UseTemplate {`⟨*type*⟩`}{`⟨*tname*⟩`}{`⟨*keyval*⟩`}`

Execute a template ⟨*tname*⟩ of type ⟨*type*⟩ at run-time using ⟨*keyvals*⟩ as the value assignments for its keys. In this case the keys are evaluated at run-time thus this method is far slower than using a predeclared instance of this template (see below). This command can also appear as the value for a key of type 'i' in which case the evaluation happens at declaration time of the template that contains this key!

`\UseInstance {`⟨*type*⟩`}{`⟨*iname*⟩`}`

Run the instance ⟨*iname*⟩ of template type ⟨*type*⟩. If a collection is in force see if there is a collection instance of name ⟨*iname*⟩ and if so run that instead.

`\UseCollection {`⟨*type*⟩`}{`⟨*collection*⟩`}`

Declare that from now on (normal scoping rules) the collection ⟨*collection*⟩ for template type ⟨*type*⟩ is in force. This means that a call to `\UseInstance` will first check if there is a collection instance defined, and if so use that instance, otherwise use the normal instance.

2.5 Test commands

`\IfExistsInstanceTF {`⟨*type*⟩`}{`⟨*iname*⟩`}{`⟨*true*⟩`}{`⟨*false*⟩`}`

Test if for template type ⟨*type*⟩ an instance with name ⟨*iname*⟩ exists. Select ⟨*true*⟩ or ⟨*false*⟩ code accordingly.

3 Examples of template key types

The general syntax for key specification in templates (fourth argument of the command `\DeclareTemplate`) is:

```
{
  ⟨key-name1⟩ =⟨key-type1⟩ ⟨optional-default1⟩ ⟨storage-bin1⟩,
  ⟨key-name2⟩ =⟨key-type2⟩ ⟨optional-default2⟩ ⟨storage-bin2⟩,
     ...
}
```

In this section we look at all possible key types and give examples for them.

3.1 Attributes that receive names as values

The type `n` expects to receive a LaTeX name as a value. Used, for example, to specify the name of a LaTeX counter to use.

```
heading-id     =n                                    \heading@id,
counter-id     =n  [\DelayEvaluation{\heading@id}] \heading@counter,
```

Notice the use of `\DelayEvaluation` in the default of `counter-id`. It is necessary to make the default the token `\heading@id` if we want to inherit the value from the `heading-id` key. Otherwise it gets value of `\heading@id` at the time the instance is declared.

3.2 Attributes that receive functions as values

The type `f`⟨*num*⟩ expects a function with ⟨*num*⟩ arguments as a value. The arguments are denoted by `#1`, `#2`, etc. In most cases either `f0` (for declarations) or `f1` (to format one argument) are needed.

```
initial-font   =f0           \initial@font,
initial-format =f1 [#1]      \initial@boxhandling,
```

3.3 Attributes that receive dimensions as values

As far as specifying instances the `l` and `L` type behave identically. They differ only in the type of internal storage-bin they need: `l` expects a length register while `L` expects an ordinary macro name and assigns its value via `\cs_set_nopar:Npn` .

```
pre-sep        =l   \topsep,
post-sep       =L   \botsep,
```

3.4 Attributes that receive integers as values

The `c` and `C` type receive integers as values. Again either of them can be transparently used. In case of `c` the ⟨*storage-bin*⟩ has to be a TEX count register not a LATEX counter name, i.e., set up via `\newcount`. (LATEX counters can be used as well if they are accessed via their internal name, i.e., via `\c@`⟨*LATEX-counter*⟩)

```
pre-penalty    =c   \@beginparpenalty,
penalty        =C   \hmaterial@penalty,
```

3.5 Attributes that receive template instances as values

The type `i{`⟨*type*⟩`}` takes as value the name of a declared instance of that type. The ⟨*storage-bin*⟩ associated with the key will store a command essentially equivalent to a call to `\UseInstance{`⟨*type*⟩`}{`⟨*name*⟩`}`, but in a slightly optimised internal form.

As an exception to this rule the replacement code may be of the form `\UseTemplate` followed by the key settings for the template but without the mandatory arguments. In this case the 'inner' instance declaration is 'pre compiled' and the token assigned to the store the value assigned to this key will execute an instance of the template directly, it will not re-parse the keyword settings each time the instance is used.

```
justification-setup =i{justification} \list@justification,
```

Usage within an instance declaration is either

```
justification-setup = raggedright,
```

i.e., name of a declared instance or a call to `\UseTemplate`

```
justification-setup = \UseTemplate{justification}{TeX}
                          { startskip = 0pt, ... },
```

3.6 Attributes that receive true or false values

The type `s` expects the strings `true` or `false` as values. In this case the declaration has no ⟨*storage-bin*⟩. Instead the declaration consists of two brace groups containing code. Depending on the value one of the groups gets copied verbatim into the internal parameter list of the instance and gets executed at run-time at the point where `\DoParameterAssignments` is seen.

```
item-implicit-boolean  =s
    { \cs_set_nopar:Npn \item@implicit@code{\item\relax} }{},
numbered-boolean       =b [true] @heading@nums,
```

3.7 Attributes that accept any value

The type `g` is a low-level specification which contains arbitrary code in place of the ⟨*storage-bin*⟩. This code is evaluated at declaration time of the instance and by default *nothing* is passed to the internal parameter list (this has to happen explicitly from within the code). `#1` may be used to access the value specified.

The main purpose for this type is of historical nature (originally most of the other types have been implemented internally using `g`).

The type `x` also requires code in place of the ⟨*storage-bin*⟩. However with this type all of the code is copied unevaluated to the internal parameter list. There are some applications for this type when implementing customisable defaults. However, it is likely that it will not survive a final release.

```
generic-key   =g \typeout{#1},
extra-assigns =x \typeout{#1},
```

4 A complete example

The following example shows a sketch of a template for typesetting captions to be used as part of a larger mechanism setting whole floats.[3]

We declare a template type `caption` then define an example template for that type and finally produce some instances from that.

[3] I made it up while I went along so if you spot the "missing brace" or some other blunder tell me, FMi.

4.1 Declaring the template type

To define the template type we first have to ask ourselves what information would be varying each time such a template is used? A potential answer could be the following:

- The float name, e.g., 'Table' or 'Fig.' etc.
- The float number e.g., '10' or '3–c' etc.
- The actual caption text as specified in the document.

Since the above items would be differed in each instantiation of such a template we would pass them as mandatory argument to the template.

Are there others? Possibly. Here are two more that seem to be useful, at least in a number of cases:

- The text of the legend in document classes that distinguish between caption text (heading to the figure/table) and legend (explanatory material)
- Measure to which the caption should be typeset.

The last one of these might need some extra explanation. Suppose a design requires that the caption width is decided depending on the width of the table of figure, e.g., the caption is supposed to typeset below some illustration and should not be wider than that illustration, or the caption is typeset aside to the illustration using the remaining space. In that case the process that formats the whole float needs to communicate with the current template to pass that (varying) information along. Of course, that could happen by using global variables, e.g., the outer process sets the measure as desired before calling the caption formatting template. What makes more sense is likely to be a matter of taste but it also has to do with the precise semantics of the template type. Staying with our example: if the the semantics of the template type `caption` is supposed to produce a formatted box (in TEX terms) then we should pass the measure as an argument if we ever intend to allow for variations. If on the other hand the semantics are to format a certain set of text into the current galley (which has measure of its own) then a measure argument would not belong to this template type.

Are there other variations sensible? Yes, for example, instead of passing a fixed string like "Fig." as the first argument one could pass an abstract float type identifier and let the template worry to deduce from that information what fixed string to produce.

Another question: why should we pass the fixed text (or an abstract identifier from which it can be deduced) and the number as separate arguments to the template instead of passing a combined string (like it is done in the `\@makecaption` command of LATEX 2_ε)? Answer: because this allows to build templates that can individually manipulate both bits of information, e.g., to format the number in a different font, etc.

So what are the conclusions of this discussion? Defining the semantics of a template type is difficult and often needs several trials to come up with something that is covering the

anticipated use. There is clearly not a cardinal way for defining template types; how the overall separation into smaller units is done is partly a matter of taste and partly a matter of the major layout characteristics that one tries to support.

Returning to our example: let's assume we settle for the first four arguments, i.e., the calling template is responsible for setting the measure for the caption text if necessary.

What we also have to do is to define (at least for ourselves) what data the arguments accept and what their semantics are. An informal summary of that could be the following:

Arg	*Data Type*	*Description*
1	text	fixed float description
2	text/`\NoValue`	float number
3	text	caption text
4	text/`\NoValue`	legend text

The second and the fourth argument are allowed to be missing (i.e., can get `\NoValue` passed as a value). Note that the empty string in case of a text argument is different from `\NoValue`.

We further declare that it is permissible for a template of this type to ignore the information provided by all arguments except 3, i.e., the caption text.

Finally the result of the template formatting should is to typeset text into a current galley (paragraph mode in LaTeX lingua).

All the above is semantic information that (at least right now) is not being enforced by declaring a template type (except for the number of arguments) but each template of a certain type is supposed to conform to this specification nonetheless.[4]

This finally leads to the following declaration:

```
\DeclareTemplateType{caption}{4}
```

4.2 Defining a first template

We start by defining a simple template of type `caption` which roughly formats a caption like those being presented in LaTeX 2_ε's article class, i.e., the caption is typeset as a paragraph if it is longer than a single line, otherwise it is centered. The legend even if present is ignored. Above and below we give the designer the possibility to add some space.

In fact the examples is more or less identical in code to `\@makecaption` except that if the second argument (i.e., the number) is `\NoValue` it and its preceding space[5] gets ignored.

[4] To make this even clearer we are thinking of extending the template type declaration with another argument in which one has to formally or informally (?) specifies information like the one in the table above.

[5] For those who wonder: spaces are by default ignored within definitions when the new packages are used due to a command `\InternalSyntaxOn`, do get a normal space one has to use `~` and to obtain an unbreakable space `\nobreakspace`.

We start by declaring the template `toosimple` of type `caption` having four mandatory arguments (as described in the discussion of the template type).

```
\DeclareTemplate{caption}{toosimple}{4}
```

The next argument of `\DeclareTemplate` lists all keys for the template. In this case we have keys for the vertical spaces above and below. We make them type `L` to save on registers but with a bit of care we could also have used scratch registers like `\@tempskipa` etc. Their default values are both zero.

```
{
  above-skip =L [Opt] \caption@above@skip ,
  below-skip =L [Opt] \caption@below@skip ,
}
```

The final argument of `\DeclareTemplate` contains the actual processing code. We start with looking at the second mandatory argument (caption number) to find out if it is `\NoValue` and depending on the result define a helper command `\caption@start`.

```
{
  \IfNoValueTF{#2}
     { \cs_set_nopar:Npn \caption@start{#1:~} }
     { \cs_set_nopar:Npn \caption@start{#1~#2:~} }
```

Having dealt with the prelims we now run `\DoParameterAssigments` at which point the keys of the template are made available, e.g., at this point all those right hand containers such as `\caption@above@skip` get assigned the value specified in an instantiation of the template. (That scheme allows to do preliminary processing up front, e.g., defaults for the keys could be assigned prior to that point in which case they are overwritten if the template instance specifies a different value. the use of specifying defaults via the `[..]` syntax as done above is slightly faster at run-time but needs more memory.)

```
  \DoParameterAssigments
```

The rest of the code should look familiar to anybody who ever looked at `article.cls`. The only point worth mentioning are the `\relax` commands after `\caption@above@skip` and `\caption@below@skip`. Since we have decided to use `L` as key type these commands are macros and not registers containing the dimensions as strings. This means that we have to be careful to ensure that TEX knows where the dimension ends. In certain cases text following such a command might be mistaken as being part of the dimension (e.g., if followed by the word `plus`, etc.). In the code below this could only happen for the second `\vskip` but it is good practice to always add a terminating `\relax` to avoid such hidden traps.

```
  \vskip \caption@above@skip \relax
  \sbox \@tempboxa {\caption@start #3}
```

```
    \ifdim \wd\@tempboxa >\hsize
      \caption@start #3\par
    \else
      \global \@minipagefalse
      \hb@xt@\hsize{\hfil\box\@tempboxa\hfil}
    \fi
    \vskip \caption@below@skip \relax
  }
```

Why is the above template of not much use? Simply because it doesn't offer any flexibility to declare different designs. The only alteration offered to the designer is to modify the space above and below the caption, e.g., the following declaration would mimic the definition within the `article.cls` class of LaTeX 2_ε:

```
\DeclareInstance{caption}{article}{toosimple}
  {
    above-skip =  0pt,
    below-skip = 10pt,
  }
```

And that's all that can be manipulated. All items that people asking to change, e.g., not having a colon after the number, using different fonts and font sizes, etc. are still hard-wired and thus inaccessible. So we have to do better if we want to make use of the power the template mechanism offers.

4.3 Defining a better template

First step in defining better templates is to ask ourselves a couple of questions:

- What are the main characteristics of the layout the template is supposed to support?
- What are the elements that we want to allow (or can allow) the designer to modify?

Take the first question first: the layout supported by the template of the previous section had as its main characteristics that it would center the caption if it would fit in a single line in the current measure. We could consider this being an unchangable characteristic of the layout this template produces (and a designer would need to use a different template of type `caption` if a design compatible with this restriction is desired) or we could try to make our template smarter by adding bells and wistles that allow the designer to say stuff like:

```
  one-line-format = \hfil #1 \hfil,
```

or

```
  one-line-action = center,
```

depending on how we intend to offer changing the behavior of the template. Like when trying to define sensible template types we have no single road to heaven (and probably as many to hell) — it has a lot to do with how we think about design.

My advice, after having tried to work with these concepts for a while, is to keep templates simple in so far as that most of not all attribute for a template should be relevant for the design. In other words, if you have attributes that, depending on their setting, make half of the other attributes not applicable then it may be appropriate to think about providing several templates instead. To give an example from LaTeX 2_ε: instead of having `\@startsection` deal with both vertical heads and run-in heads provide individual templates. (`\@startsection` is this famous command where design switches are build in by making dimensions negative to signal something and afterwards use the absolute value.) Another way to look at this is to say that a template should normally not contain large amounts of code which is only selected in a subset of attribute settings.

As said before there are no golden rules, it is perfectly possible to make hugely complicated templates that solve every possible aspect of layout one could think of in one go — it is just that with keeping it more simple one can get the same functionality with less headaches for the template writer as well as the template user later on.

Returning to our example: allowing to handle the case of a single line caption specially could well be considered part of the template. In contrast: layouts that would put the caption number sideways, i.e., which would need totally different internal coding should probably be coded as a separate template of type `caption`.

So for our next example template we settle for the fixed caption text plus number (if any) being at the beginning of the variable caption text (coming from the document) and being together formatted as some sort of a pargraph. In case of the whole caption being a single line we allow the designer to specify how to lay it out (e.g., centered, flush left, etc.). If there is a legend it will get formatted by a vertical space followed by the legend formatted as another paragraph.

More precisely we allow for the following bells and wistles:

```
\DeclareTemplate{caption}{lesssimple}{4}
  {
```

The designer can specify the space above and below the caption like we did in our first example.

```
   above-skip         =L           [0pt] \caption@above@skip ,
   below-skip         =L           [0pt] \caption@below@skip ,
```

Regarding the caption number we support the case where no number is present (the value being `\NoValue`) as well as the number being present. For both cases the designer has to specify what formatting should be attached. By default all is being typeset in the font the whole caption is presented but if there is a need for it the designer can use the following

keys to attach special formatting devices to each particular item beside specifying special spacing information or replacing the default colon after the number with something else.

```
number-format      =f2      [#1~#2:~] \caption@number@format,
nonumber-format    =f1         [#1:~] \caption@nonumber@format,
```

If the caption is fitting onto a single line we make it possible for the designer to specify how this single line should be positioned (the default is to center the line).

```
single-line-format =f1 [\hfil#1\hfil] \caption@single@line@format,
```

The font for the caption (including the fixed text and the number unless specified differently above) is going to be the one decided by the next key.

```
caption-font       =f0  [\normalfont] \caption@font,
```

The next attribute deserves some extra explanation: here we make use of an interface which is explained in more detail when we reveil the support for galley formatting.[6] In a nutshell the template type `hj` (hyphenation & justification) allows one to define a) the justification concepts applied to the upcoming paragraphs, e.g., whether they should be set flush left, adjusted, first line centered, etc. b) the linebreaking strategy used and c) the hyphenation rules which should apply. All this is done by selecting an appropriate (predefined) instance of this type as will hopefully become somewhat clearer in the example instances shown below.

```
caption-hj-setup   =i {hj}  [default] \caption@hj@instance,
```

In case there is a legend to format we give the designer the possibility to specify by how much vertical space it should be separated from the preceding paragraph (i.e., the caption text). The attributes for font and hj setup are comparable to those for the caption text itself (except that they will only apply to the legend). The only addition is the key `legend-text` which is allowed to take a fixed text (plus any formating and spacing for it) which will be added to the front of the legend in case it is provided at all (by default it is empty).

```
  legend-sep         =L           [0pt] \caption@legend@sep ,
  legend-text        =f0             [] \caption@legend@text,
  legend-font        =f0  [\normalfont] \caption@legend@font,
  legend-hj-setup    =i {hj}  [default] \caption@legend@hj@instance,
}
```

[6] Guess I have to apologize for the fact that i partly make use of that interface in this example while on other occasions (like the use of vertical spacing) within the example I do not—consistency around midnight is not my strength I fear (FMi).

The actual code for the template should hold few if any surprises. In fact it is more or less identical to the one of the first template example, except that now we have now taken out some of the hardwired decisions and placed them into attributes.

```
{
  \IfNoValueTF{#2}
     { \cs_set_nopar:Npn \caption@start{\caption@number@format{#1}{#2}} }
     { \cs_set_nopar:Npn \caption@start{\caption@nonumber@format{#1}}   }
  \DoParameterAssigments
  \vskip \caption@above@skip \relax
```

To properly measure the caption to determine if it fits a single line we have to set it in the right font, so here as well as below we have to apply `\caption@font`.

```
  \sbox \@tempboxa {\caption@font \caption@start #3}
  \ifdim \wd\@tempboxa >\hsize
    \begingroup
      \caption@font \caption@hj@instance
      \caption@start #3\par
    \endgroup
  \else
    \global \@minipagefalse
    \hb@xt@\hsize{\caption@single@line@format{\box\@tempboxa}}
  \fi
```

To decide whether or not we have to set any legend we have to test `#4` for being `\NoValue`. This part of the code was not present in the previous example but otherwise should be straight forward.

```
  \IfNoValueF{#4}
    {
     \vskip \caption@legend@sep \relax
     \begingroup
        \caption@legend@font \caption@legend@hj@instance
        \caption@legend@text
        #4\par
     \endgroup
    }
  \vskip \caption@below@skip \relax
}
```

I wouldn't claim the the above template is good or contains everything that would be desired and I'm sure that in the end we will have several such template for typesetting the caption part and perhaps decide on a different template type in the first place. So this is only to give a glimpse of how the template interface could be applied and I hope that reading it can see a) how they can apply it to other areas as well as see what is wrong with the example itself.

To just note one point that i thought of being wrong after writing the above paragraphs: the key `single-line-format` was declared to be a function with one argument with the idea that besides specifying the single line should be centered (`\hfil`) on both sides, or flush left, or flush right (`\hfil` on one side) one could also specify something like

```
single-line-format = \hspace{10pt}#1\hfil,
```

that is a fixed indentation on the left in case where the caption is a single line. However, of course one can't. Or at least it is not safe to do so since our test in the code tests the width of the line without taking into account such a finite fixed space and guess what might happen? So in summary, flexibility needs some thought and often some afterthoughts as well — happy thinking :-)

4.4 Defining a few instances

So let us conclude this example with a few sample instances. We start with one that repeats what current LaTeX 2_ε provides in the article class. It shows all keys with values. However in fact only the first key is actually needed since all others are the same as the default values in the template (and of course a legend is not specifiable in standard LaTeX coding so those settings simply do not apply anyway).

```
\DeclareInstance{caption}{article}{lesssimple}
  {
   above-skip         = 10pt,
   below-skip         = 0pt,
   number-format      = #1~#2:~,
   nonumber-format    = #1:~,
   single-line-format = \hfil#1\hfil,
   caption-font       = \normalfont,
   caption-hj-setup   = default,
   legend-sep         = 0pt,
   legend-text        = ,
   legend-font        = \normalfont,
   legend-hj-setup    = default,
  }
```

The next examples are taken from books on the shelf essentially a random selection I fear. This one is from *Introduction to Database Design* by C. J. Date and it uses Helvetica for the caption text with the caption flush left, with the figure and the fixed string (e.g., 'Fig.' in bold face) separated by a quad of space. No legend either so this is not set up. The `hj` instance `noindentflushleft` is supposed to produce a ragged right paragraph without any indentation. It would have to be set up elsewhere (instance to the template of type `hj`).

```
\DeclareInstance{caption}{DATE}{lesssimple}
  {
```

```
   above-skip         = 10pt,
   below-skip         = 0pt,
   number-format      = \textbf{#1~#2}\quad,
   nonumber-format    = \textbf{#1}\quad,
   single-line-format = #1\hfil,
   caption-font       = \fontfamily{phv} \normalfont,
   caption-hj-setup   = noindentflushleft,
  }
```

The final example is from the book "Methods of Book Design" by H. Williamson which sets the caption centered if it fits a single line but adjusted as a paragraph without any indentation if longer than a single line. It uses old style numerals followed by a period for the number (though the example isn't quite right as i guess the text font used already has oldstyle numerals as default, so `\oldstylenums` is in fact not necessary).

```
\DeclareInstance{caption}{WILLIAMSON}{lesssimple}
  {
   above-skip         = 10pt,
   below-skip         = 0pt,
   number-format      = #1~\oldstylenums{#2}.~,
   nonumber-format    = #1~,
   single-line-format = \hfil#1\hfil,
   caption-font       = \normalfont,
   caption-hj-setup   = noindentadjusted,
  }
```

5 Notes

5.1 Note on multi-valued parameters

The following code[7] implements for registers (ie L,l,C,c keys) and for names (ie n key) a multi-selection mechanism of the following form:

```
    key    = \MultiSelection \ListDepth {
                    \DelayEvaluation {2.5em},
                    20pt + 34pt }
                  { \DelayEvaluation {1em} },
```

where the first argument to `\MultiSelection` is a counter, the second argument is a comma separated list of values denoting the values for the cases 1, 2,..., and the third argument contains the value for all other cases.

The values are evaluated at declaration time in case of registers and therefore can contain calc expressions as well as `\DelayEvaluation`.

[7] docu taken from trial implementation in xlists.dtx, FMi

Due to the implementation the case list is not allowed to have a trailing comma! And of course no checks are made whatsoever :-(

A probably much nicer syntax would be something like this:

```
key    = \MultiSelection {
             selector = \ListDepth,
             1        = \DelayEvaluation {2.5em},
             2        = 20pt + 34pt,
             else     = \DelayEvaluation {1em}
           },
```

but i found that too difficult to implement right now.

I think it should also be considered if this kind of thing should be a generally available feature on all key types especially on the f⟨*number*⟩ ones.

Anyway it is what i need for lists right now and as such it is sufficient.

5.2 Notes on template restriction

Possible semantics:

a: just:-) changes the defaults ie the new template has as defaults those of its source as modified by the supplied keyvals;

b: similar to a: but also removes some keywords ie the new template will not accept the keywods whose values are set by the suppied keyvals;

c: plan C.

Towards an implementation of b: but without a restriction on what keys appear where.

5.3 Open issues

In this section unresolved issues or ideas to think about and perhaps implement are collected. There is no particular order to them.

- The order of arguments in `\UseCollection` is illogical in my eyes! A collection typically modifies the behavior of several types and thus should perhaps be first (as it is in the `\DeclareCollectionInstance` case). Or not, or what?
- How should `\IfExistsInstanceTF` behave for Collection instances? Do we need a special check for those or a default action? Or do we need an additional test for the existence of collection instances?

- It was suggested that the template type declaration should get another argument in which (in?)formally the semantics for the template types are described, e.g., data type of arguments, resulting output, ... (somewhat like the description arguments for functions and variables in Emacs-Lisp). The advantage being that this helps employing the templates better as well as perhaps guiding context sensitive editors to support the work with such templates (e.g., providing help texts).
- The same might be of interest for the keys of individual templates though here syntax support is already available to some extend by the declaration of key types.
- There might be a need to distinguish between TEX's dimen and skip registers. Right now this is not done and both `l` and `L` accepts what LATEX calls "rubber length" specifications.
- The type `b` can probably vanish. It is equivalent to specifying the mutators of a `\newif` command in the brace groups, e.g.

```
numbered-boolean =b [true] @heading@nums,
numbered-boolean =s [true] {\@heading@numstrue}
                           {\@heading@numsfalse},
```

- See issue raised about syntax (and semantics) for `\Multiselection`.
- `f0` keys should perhaps support `\UseTemplate` by replacing it with its internal form. or perhaps this is a rubbish idea?
- Marcin Wolinski suggested to use `\EvalOnUse` instead or in addition to `\DelayEvalutation`.

6 Implementation

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\RequirePackage{expl3}
\RequirePackage{ldcsetup,xparse}
```

Declare a private token register for building parameter lists. Having the number saves a few expandafters (probably not needed in the end).

```
\toks_new:N\l_TP_KV_assignments_toks
\toks_new:N\l_TP_default_assignments_toks
```

`\TP_declare_instance:Nnn`
`\TP_declare_instance:cnn`

Declare a command name to be an instance of a template ie with a particular setting of the parameters.
`#1` internal command name for instance to be (globally) declared
`#2` template type/template name
`#3` key value assignments for parameters of `#2`

```
\cs_new_nopar:Npn \TP_declare_instance:Nnn #1#2#3 {
  \group_begin:
    \TP_instdecl_generate_assignments:nn {#2}{#3}
    \cs_gset_nopar:Npx #1 {
      \tl_if_eq:cNTF { TP>/#2 } \c_TP_doparameterassignments_tl
```

If the body of the template consists only of the token `\DoParameterAssignments`, then we insert the list of parameter assignments directly. Otherwise we have push them onto the stack and prepare to execute the body code (which in turn will pop them again when it reaches `\DoParameterAssignments` inside).

```
          { \toks_use:N \l_TP_KV_assignments_toks }
          {
            \exp_not:N \TP_push_assignments:n
               {\toks_use:N\l_TP_KV_assignments_toks}
             \exp_not:c {TP>/#2}
          }
       }
  \group_end:}
\cs_generate_variant:Nn \TP_declare_instance:Nnn {cnn}
```

`\c_TP_doparameterassignments_tl`

```
\tl_set:Nn \c_TP_doparameterassignments_tl {\DoParameterAssignments}
```

`\UseTemplate` `{type}{templatename}{keyval}` Directly use a template with a particular parameter setting. This is also picked up if used in a nested fashion inside a parameter list.
`#1` type of a template.
`#2` name of a template.
`#3` key value assignments for parameters of `#1`.

```
\cs_new_nopar:Npn \UseTemplate #1#2#3{
  \TP_instdecl_generate_assignments:nn {#1/#2}{#3}
  \TP_push_assignments:
  \use:c { TP>/#1/#2 }
}
```

`\DoParameterAssignments` Access the parameter assignment list that was once stored in `\l_TP_KV_assignments_toks` and then moved onto the `\g_TP_assignments_stack_toks`.

```
\cs_new_nopar:Npn \DoParameterAssignments{
  \exp_after:wN
    \TP_pop_and_execute_assignments:nw
       \toks_use:N \g_TP_assignments_stack_toks \q_stop
}
```

`_pop_and_execute_assignments:nw`

```
\cs_new_nopar:Npn \TP_pop_and_execute_assignments:nw#1#2\q_stop{
  \toks_gset:Nn \g_TP_assignments_stack_toks {#2}
  #1}
```

\g_TP_assignments_stack_toks

```
\toks_new:N   \g_TP_assignments_stack_toks
\toks_gset:Nn \g_TP_assignments_stack_toks {\scan_stop:}% avoid brace loss
```

\TP_push_assignments:n
\TP_push_assignments:

Push a list of parameter assignments onto the \g_TP_assignments_stack_toks. As it all happens in token registers #s need no doubling. \TP_push_assignments: expects it to be \l_TP_KV_assignments_toks (needs fixing).

```
\cs_new:Npn \TP_push_assignments:n#1{
  \toks_gput_left:Nn\g_TP_assignments_stack_toks{{#1}}
}
\cs_new_nopar:Npn \TP_push_assignments:{
  \toks_gset:No \g_TP_assignments_stack_toks
    {\exp_after:wN
     {\toks_use:N\exp_after:wN\l_TP_KV_assignments_toks\exp_after:wN}
      \toks_use:N\g_TP_assignments_stack_toks}
}
```

\DeclareTemplateType {type}{nofarg}

```
\cs_new_nopar:Npn \DeclareTemplateType #1#2{
  \tl_set:cn {TP@<#1>} {{}#2}}
```

\TP_get_csname_prefix:n {type} returns prefix for csnames for template type, based on current collection.

```
\cs_new_nopar:Npn \TP_get_csname_prefix:n #1 {
  <\exp_after:wN\exp_after:wN\exp_after:wN
   \use_i:nn
      \cs:w TP@<#1>\cs_end:>#1/
}
```

\TP_get_arg_count:n {type} returns arg count for the template type.

```
\cs_new_nopar:Npn \TP_get_arg_count:n #1 {
  \exp_after:wN\exp_after:wN\exp_after:wN
  \use_ii:nn
    \cs:w TP@<#1>\cs_end:
}
```

\DeclareTemplate {type}{templatename}{nofarg}{keywordspec}{code}

```
\cs_new:Npn\DeclareTemplate #1#2#3#4#5{
  \cs_if_free:cTF{TP@<#1>}
    {\undefinedtype\DeclareTemplateType{#1}#3}
    {
      \intexpr_compare:nNnF{#3}={\TP_get_arg_count:n{#1}}
      { \BadArgCount }
    }
```

Parse the key declaration, and execute the list with a suitable definition of `\KV@elt`.

```
\cs_set_eq:NN \KV_elt:nn \TP_templdecl_process_KV:nn
\cs_set_nopar:Npn \KV_default_elt:n##1{
  \PackageError{template}{Missing~ =~ after~ ##1}\@ehd}
\cs_set_eq:NN\KV@elt\KV_elt:nn
\cs_set_eq:NN\KV@default@elt\KV_default_elt:n
\tl_set:Nn \l_TP_curr_name_tl {#1/#2}
\toks_clear:N\l_TP_default_assignments_toks
```

At this point there should be a test for which catcode regime we are in. We just test if spaces are ignored.

```
%\intexpr_compare:nNnTF{\char_value_catcode:n{`\ }}=\c_nine
%\KV_parse_picky_no_space_removal_no_sanitize:n
%\KV_parse_picky_space_removal_no_sanitize:n
\KV@parse{#4}
```

Define the defaults: the setting for `TPD>/\l_TP_curr_name_tl` is a tricky since `\l_TP_default_assignments_toks` may contain `#`. We have to use an `x` expansion rather than `o` since that will hide those during the assignment. FIX THIS (see below)!

```
\cs_set_nopar:cpx { TPD>/\l_TP_curr_name_tl }
            {\toks_use:N\l_TP_default_assignments_toks}

\tl_clear:c {TPR>/\l_TP_curr_name_tl}

\tl_set_eq:cN {TPO>/\l_TP_curr_name_tl}\l_TP_curr_name_tl
```

Define the template (using `\cs_new_nopar:Npn` means that one can't redefine a template easily).

```
\cs_generate_from_arg_count:cNnn {TP>/\l_TP_curr_name_tl}
\cs_set:Npn {#3}{#5}
}
```

`\tl_set:cx` FIX this: The code above only works because in the past `\tl_set:cx` was defined as follows:

```
\cs_set:Npn \tl_set:cx {\exp_args:Nc  \tl_set:Nx}
```

i.e., expanding the second arg at the very end. This is no longer the case but for the moment I revert to the old definition until the template code is fixed to not rely on `\tl_set:cx` expanding the second arg at the very last minute.

`\TP_templdecl_process_KV:nn` The list of undefined keys and values is put in the list of the form
`\KV_elt:nn{`⟨*key1*⟩`}{`⟨*val1*⟩`}\KV_elt:nn{`⟨*key2*⟩`}{`⟨*val2*⟩`}`...
So just need to give this macro a suitable definition. We just need to look at the first token of the value, to see what sort of key it is, so call a helper function to split that off.

```
\cs_new_nopar:Npn \TP_templdecl_process_KV:nn #1#2 {%
  \cs_set_eq:NN \TP_templdecl_add_global_or_nothing: \prg_do_nothing:
  \bool_set_false:N\l_TP_global_assignment_bool
  \tl_set:Nn\l_TP_currkey_tl{#1}
  \TP_templdecl_parse_KV:N#2\q_stop}
```

`\TP_templdecl_parse_KV:N` Case switch on the possible key types.

```
\cs_new_nopar:Npn \TP_templdecl_parse_KV:N #1 {
```

In `#1` we have key, in `#2` the first character after the equal sign and in `#3` the remainder of the line. We now have to parse that remainder to find out if it contains a default value (in brackets) and then set up the key declaration needed to parse instance declarations. The method is similar in most cases: we call `\TP_parse_optional_key_default:nw` which parses for the default and pass it already found key name as first argument, what to do in the end as second argument, and the remainder delimited by `\q_stop` so that it becomes parseable.

Note that the code in the second argument to `\TP_parse_optional_key_default:nw` normally calls on a macro with one more argument than actually provided: the reason being that the missing argument will be the remainder of the line (added by `\TP_parse_optional_key_default:nw` after the default has be removed (if present)).

```
  \cs_if_free:cTF{TP_use_arg_type_#1:w}
  {\PackageError{template}{Unknown~key~type~ (#1)~for~\l_TP_currkey_tl}\@eha}
  {\use:c{TP_use_arg_type_#1:w}}
```

The `f` and `i` keys are somewhat different since there we first have to parse for an additional argument (a digit in case of `f` or an template type name in case of `i`):

One more alternative: a `+` after the equal sign signals global so we change `\TP_templdecl_add_global_or_n` to append a `\pref_global:D` to the assignment toks and then reparse the rest.

```
%     \cs_set_nopar:Npn  \TP_templdecl_add_global_or_nothing:
%          {\toks_put_right:Nn \l_TP_KV_assignments_toks {\pref_global:D} }
%     \TP_templdecl_parse_KV:nw{#1}#3\TP_templdecl_parse_KV:nnw
}
```

`\l_TP_global_assignment_bool` For keeping track of the assignments.

```
\bool_new:N \l_TP_global_assignment_bool
```

`\TP_use_arg_type_+:w` The `+` does two things: Sets a boolean true to be used by the types that can't simply be prefixed with `\pref_global:D` and defines `\TP_templdecl_add_global_or_nothing:` to put the prefix onto the list. After that we simply call the big switch again.[8].

```
\cs_new_nopar:cpn{TP_use_arg_type_+:w} {
```

[8] It should probably all be changed to not rely on the prefix working

```
  \bool_set_true:N\l_TP_global_assignment_bool
  \cs_set_nopar:Npn  \TP_templdecl_add_global_or_nothing:
  {\toks_put_right:Nn \l_TP_KV_assignments_toks {\pref_global:D} }
  \TP_templdecl_parse_KV:N
}
```

`\TP_use_arg_type_l:w` The `l` sets a length register. We disable the prefix and insert either `\gsetlength` or `\setlength` depending on whether a `+` was seen or not.

```
\cs_new_nopar:Npn\TP_use_arg_type_l:w {
  \TP_parse_optional_key_default:nw
  {
    \cs_set_eq:NN \TP_templdecl_add_global_or_nothing: \prg_do_nothing:
    \bool_if:NTF \l_TP_global_assignment_bool
    {\TP_templdecl_setup_register_key:Nn\gsetlength}
    {\TP_templdecl_setup_register_key:Nn\setlength}
  }
}
```

`\TP_use_arg_type_L:w` The `L` sets a fake length register.

```
\cs_new_nopar:Npn\TP_use_arg_type_L:w {
  \TP_parse_optional_key_default:nw
  {\TP_templdecl_setup_fakeregister_key:NNn\setlength\l_tmpa_skip}
}
```

`\TP_use_arg_type_c:w` The `c` sets a count register.

```
\cs_new_nopar:Npn\TP_use_arg_type_c:w {
  \TP_parse_optional_key_default:nw
  {
    \cs_set_eq:NN\TP_templdecl_add_global_or_nothing:\prg_do_nothing:
    \bool_if:NTF \l_TP_global_assignment_bool
    {\TP_templdecl_setup_register_key:Nn\GSetInternalCounter}
    {\TP_templdecl_setup_register_key:Nn\SetInternalCounter}
  }
}
```

`\TP_use_arg_type_C:w` The `C` sets a fake count register.

```
\cs_new_nopar:Npn\TP_use_arg_type_C:w {
  \TP_parse_optional_key_default:nw
  {\TP_templdecl_setup_fakeregister_key:NNn
    \SetInternalCounter\l_tmpa_int}
}
```

`\TP_use_arg_type_n:w` The `n` sets a token list variable.

```
\cs_new_nopar:Npn\TP_use_arg_type_n:w {
```

```
  \TP_parse_optional_key_default:nw
  {\TP_templdecl_setup_n_key:N}
}
```

`\TP_use_arg_type_f:w` `\TP_templdecl_parse_f_arg:nw`

The `f` defines a command with between 0 and 9 arguments.

```
\cs_new_nopar:Npn\TP_use_arg_type_f:w #1{
  %\TP_templdecl_parse_f_arg:nw {#1}
  \TP_parse_optional_key_default:nw{\TP_templdecl_setup_f_key:Nn{#1}}
}
```

Helper for `\TP_templdecl_parse_KV:nnw`.

```
\cs_new_nopar:Npn \TP_templdecl_parse_f_arg:nw#1#2{
```

The third argument of `\TP_templdecl_setup_f_key:Nn`, i.e., the macro name, is the remaining data up to `\q_stop` which is picked up by `\TP_parse_optional_key_default:nw`.

```
  \TP_parse_optional_key_default:nw{\TP_templdecl_setup_f_key:Nn{#1}{#2}}
}
```

`\TP_use_arg_type_b:w` `\TP_templdecl_setup_b_key:nn`

The `b` uses access to the `\if_true:` and `\if_false:` primitives. Needed?

```
\cs_new_nopar:Npn\TP_use_arg_type_b:w {
  \TP_parse_optional_key_default:nw
  {\TP_templdecl_setup_b_key:n}
}

\cs_new_nopar:Npn \TP_templdecl_setup_b_key:n#1{
  \cs_set_eq:cN { if#1 } \if_true:
  \TP_templdecl_define_key:n
  { \TP_templdecl_eval_b_key_value:nn {#1}{##1} }
}
```

`P_templdecl_eval_b_key_value:nn`

Modify so the boolean does not need to have been declared with `\newif`

```
\cs_new_nopar:Npn \TP_templdecl_eval_b_key_value:nn#1#2{
  \cs_if_free:cTF {if#2}
    { \PackageError{template}{Bad~boolean~setting~#1=#2}\@eha }
    { \tl_set_eq:cc {if_#1:}{if_#2:}
      \toks_put_right:Nf \l_TP_KV_assignments_toks
         { \tl_set_eq:cc {if_#1:}{if_#2:} }
    }
}
```

`\TP_use_arg_type_s:w` `\TP_templdecl_setup_s_key:n`

The `s` chooses between true and false.

```
\cs_new_nopar:Npn\TP_use_arg_type_s:w {
  \TP_parse_optional_key_default:nw
  {\TP_templdecl_setup_s_key:n}
}
```

```
\cs_new_nopar:Npn \TP_templdecl_setup_s_key:n #1 {
  \TP_templdecl_define_key:n
    { \TP_templdecl_eval_s_key_value:nnn{##1}#1 }
}
```

`\TP_use_arg_type_i:w` The `i` expects an instance.

```
\cs_new_nopar:Npn\TP_use_arg_type_i:w #1{
  \TP_parse_optional_key_default:nw{\TP_templdecl_setup_i_key:nnn{#1}}
}
```

declaration `hd =i{head} \fooo`
use `hd = mine`
makes `\fooo` shorthand for `\UseInstance{head}{mine}`

also allowed: `hd = \UseTemplate{head}{...}{...}`
in case you want to use an unnamed instance of type `head` in this place.

`\TP_templdecl_setup_i_key:nnn` MH change: do either local or global.

```
\cs_new_nopar:Npn \TP_templdecl_setup_i_key:nnn#1#2{
  \TP_templdecl_define_key:n
  {
    \TP_templdecl_eval_i_key_value:Nnn #2 {#1}{##1}
  }
}
```

`_templdecl_eval_i_key_value:Nnn` MH change: Add extra argument for local or global.

```
\cs_new_nopar:Npn \TP_templdecl_eval_i_key_value:Nnn #1#2#3 {
  \tl_if_head_eq_meaning:nNTF {#3.}\UseTemplate
  {
    \iow_term:x{\token_to_str:N\UseTemplate\space seen}
```

Code below from `\TP_templdecl_setup_f_key:Nn` (should be combined and cleaned up) at this point one should also check if first arg of `\UseTemplate` corresponds to `#2` and if not complain (not done)

```
    {\TP_templdecl_declare_tmp_instance:nnnn #3 }
    \toks_put_right:No \l_TP_KV_assignments_toks
          { \exp_after:wN \KV@toks \exp_after:wN {\g_tmpa_tl} }
    %\TP_templdecl_add_global_or_nothing:
    %\toks_put_right:Nn \l_TP_KV_assignments_toks
    %      { \cs_set_nopar:Npx #1{ \toks_use:N \KV@toks} }
    \bool_if:NTF \l_TP_global_assignment_bool
    {\toks_put_right:Nn \l_TP_KV_assignments_toks
      {\cs_gset_nopar:Npx #1 { \toks_use:N \KV@toks}}
    }
    {\toks_put_right:Nn \l_TP_KV_assignments_toks
```

```
      {\cs_set_nopar:Npx #1 { \toks_use:N \KV@toks}}
    }
  }
  {
    \TP_let_instance:Nnn#1{#2}{#3}
```

We want the `\cs_set_eq:Nc` hiding in `\TP_let_instance:Nnn` to expand fully to two csnames so we put a `\tex_romannumeral:D 0` (which in itself expands to nothing) in front. This expands the `\cs_set_eq:Nc` fully before finding out that `\cs_set_eq:NwN` is not expandable.

```
      \toks_put_right:Nf  \l_TP_KV_assignments_toks
        { \TP_let_instance:Nnn#1{#2}{#3} }
  }
}
```

`\TP_use_arg_type_x:w` `\TP_templdecl_setup_x_key:nn`

The `x` runs internal code.

```
\cs_new_nopar:Npn\TP_use_arg_type_x:w {
  \TP_parse_optional_key_default:nw
  {\TP_templdecl_setup_x_key:n}
}

\cs_new_nopar:Npn \TP_templdecl_setup_x_key:n#1{
  \TP_templdecl_define_key:n
  { \toks_put_right:Nn\l_TP_KV_assignments_toks{#1} }
}
```

`\TP_use_arg_type_g:w` `\TP_templdecl_setup_g_key:nn`

The `g` runs any code.

```
\cs_new_nopar:Npn\TP_use_arg_type_g:w {
  \TP_parse_optional_key_default:nw
  {\TP_templdecl_setup_g_key:n}
}

\cs_new_nopar:Npn \TP_templdecl_setup_g_key:n #1 {
  \TP_templdecl_define_key:n{#1}}
```

`\TP_templdecl_define_key:n`

Here we define the key in the current template. Original code from r522 is essentially unreadable but we keep it here until the internal structure is finally sorted out.

```
\cs_new_nopar:Npn \TP_templdecl_define_key:n#1{
   \exp_after:wN  \cs_set:Npn  \cs:w
    KV@\l_TP_curr_name_tl @\l_TP_currkey_tl
     \exp_after:wN\cs_end:
     \exp_after:wN##\exp_after:wN1\exp_after:wN{
       \exp_after:wN\TP_templdecl_remove_from_default_assignments:N
       \cs:w KV@\l_TP_curr_name_tl @\l_TP_currkey_tl
       \exp_after:wN \cs_end:
```

```
        \TP_templdecl_add_global_or_nothing:
        #1
      }
}
```

`P_parse_optional_key_default:nw` Look for default value.

```
\cs_set:Npn \TP_ignore_leading_space_in_arg_ii:nn#1#2{
  \exp_args:Nf\TP_ignore_leading_space_in_arg_ii_aux:nn
  {\exp_not:N #2}{#1}
}
\cs_set:Npn \TP_ignore_leading_space_in_arg_ii_aux:nn#1#2{#2{#1}}


\DeclareDocumentCommand\TP_parse_optional_key_default:nw{mo}{
  \IfNoValueTF{#2}
    {\TP_templdecl_finish_key_setup:nw{#1}}
    {\TP_templdecl_finish_key_setup_with_default:nnw{#1}{#2}}
}
%\show\TP_parse_optional_key_default:nw
%\exp_args:Nc\show{\string\TP_parse_optional_key_default:nw}
```

`P_templdecl_finish_key_setup:nw` After having parsed the line and not found any default value it remains to actually define the key for the instance parsing by executing the setup code (in `#1`) giving it `#2` (i.e., the remainder of the line) as an argument.

```
\cs_new_nopar:Npn \TP_templdecl_finish_key_setup:nw#1#2\q_stop{
  \TP_ignore_leading_space_in_arg_ii:nn{#1}{#2}
  %%%#1{#2}
}
```

`nish_key_setup_with_default:nnw` If there is a default the situation is more complicated since we not only have to set up the key for the instance but also have to add the default value to `\l_TP_default_assignments_toks` in an appropriate way.

First set up the the key itself:

```
\cs_new_nopar:Npn \TP_templdecl_finish_key_setup_with_default:nnw#1#2#3\q_stop{
  \TP_ignore_leading_space_in_arg_ii:nn{#1}{#3}
  %%%  #1 {#3}
```

Now we run the new key code (which is stored in `\KV@...` hopefully) and give it the default found. By doing this in a group and by locally emptying `\l_TP_KV_assignments_toks` we will get the resulting assignment code into that register.

(We set `\TP_templdecl_remove_from_default_assignments:N` to `\use_none:n` since this is a temporary operation and we don't want to change the default really.)

```
  \group_begin:
    \toks_clear:N \l_TP_KV_assignments_toks
    \cs_set_eq:NN \TP_templdecl_remove_from_default_assignments:N \use_none:n
    \use:c{KV@\l_TP_curr_name_tl @\l_TP_currkey_tl}{#2}
```

And now for a final trick: before closing the group again and losing our local changes we run `\exp_after:wN` several times to get the value of `\l_TP_KV_assignments_toks` into `\l_TP_default_assignments_toks` outside that group!

```
  \exp_after:wN
  \group_end:
  \exp_after:wN
  \toks_set:Nn
  \exp_after:wN
  \l_TP_default_assignments_toks
  \exp_after:wN
    { \cs:w KV@\l_TP_curr_name_tl @\l_TP_currkey_tl \exp_after:wN \cs_end:
  \exp_after:wN
      { \toks_use:N \exp_after:wN \l_TP_KV_assignments_toks
  \exp_after:wN
      }
      \toks_use:N\l_TP_default_assignments_toks
  }
}
```

`\c_TP_true_tl`

```
\tl_new:Nn \c_TP_true_tl {true}
```

`_templdecl_eval_s_key_value:nnn`

```
\cs_new_nopar:Npn \TP_templdecl_eval_s_key_value:nnn#1#2#3 {
% no error check on this yet.
  \tl_set:Nn \l_tmpa_tl {#1}
  \tl_if_eq:NNTF \l_tmpa_tl \c_TP_true_tl
     { \toks_put_right:Nn \l_TP_KV_assignments_toks {#2} }
     { \toks_put_right:Nn \l_TP_KV_assignments_toks {#3} }
}
```

`empldecl_setup_register_key:Nnn` This is normally called automatically by `\DeclareTemplate`.

Command for setting a template attribute whose name corresponds directly to a TEX count or length register
`#1` the function to set the value eg `\setlength`.
`#2` key name.
`#3` the register to set.

This command *fully evaluates* the argument at declare time, and assigns the value to the register. It also passes an assignment of the register to the final value into the parameter list for the template.

If the value is a call to `\DelayEvaluation`, don't evaluate it now, just pass the whole assignment to the template. Remember to remove the `\DelayEvaluation`.

```
\cs_new_nopar:Npn \TP_templdecl_setup_register_key:Nn #1#2{
```

```
  \TP_templdecl_define_key:n{
    \tl_if_head_eq_meaning:nNTF{##1}\DelayEvaluation
    {
```

Old line commented out. Remove `\DelayEvaluation` and also remove the braces surrounding its argument.

```
      \toks_put_right:Nn \l_TP_KV_assignments_toks {#1#2{##1}}
      %\toks_set:No\l_tmpa_toks{\use_ii:nn ##1}
      %\toks_put_right:Nx \l_TP_KV_assignments_toks
      %  {\exp_not:n{#1#2}{\toks_use:N \l_tmpa_toks}}
    }
```

check for `\MultiSelection` creeping up and if so add something like

```
\setlength\register{\ifcase\selector \or value1 \or value2
                    ... \else valueotherwise \fi}
```

to `\l_TP_KV_assignments_toks`.

```
    {
      \tl_if_head_eq_meaning:nNTF{##1..}\MultiSelection
      {
        \group_begin:
          \TP_multiselection_add:nnnnnn #1#2##1
        \group_end:
```

there are probably better ways to do this (:-)

```
        \tl_if_in:onTF{\toks_use:N\g_TP_multiselection_toks}\DelayEvaluation
        {
          \toks_put_right:No\l_TP_KV_assignments_toks
          {
            \exp_after:wN#1\exp_after:wN#2\exp_after:wN
            {\toks_use:N\g_TP_multiselection_toks}
          }
        }
        {
          \toks_put_right:No\l_TP_KV_assignments_toks
          {
            \exp_after:wN  #2
            \exp_after:wN= \toks_use:N\g_TP_multiselection_toks\scan_stop:
          }
        }
```

otherwise do as before

```
      }
```

```
      {
        #1#2{##1}
        \toks_put_right:No\l_TP_KV_assignments_toks {
          \exp_after:wN #2 \exp_after:wN = \tex_the:D #2\scan_stop:
        }
      }
    }
  }
}
```

\DelayEvaluation
\MultiSelection

Since we are testing explicitly for `\DelayEvaluation` and `\MultiSelection` a few places we better give them unique meanings!

```
\cs_new_nopar:Npn\DelayEvaluation #1{\use_none:n{\DelayEvaluation}#1}
\cs_new_nopar:Npn\MultiSelection  #1{\use_none:n{\MultiSelection}#1}
```

move_from_default_assignments:N

Note: the toks register is more or less a plists and should perhaps be implemented as such as this would make far more readable code.

```
\cs_new_nopar:Npn \TP_templdecl_remove_from_default_assignments:N#1{
  \cs_set_nopar:Npn \TP_tmp:w ##1#1##2##3#1##4\q_stop{
    \l_TP_default_assignments_toks{##1##3}
  }
  \exp_after:wN \TP_tmp:w
     \toks_use:N\l_TP_default_assignments_toks #1\scan_stop:#1\q_stop}
```

\TP_templdecl_setup_f_key:Nn

Same for macro names. Again usually called automatically when declaring a new template.
`#1` Determines how many arguments the function should have.
`#2` The macro to be defined.

If the '`##1`', the value passed as the argument of the key to the macro `#2` is invoked starts with `\FunctionInstance`, then a special procedure is taken. Instead of defining a macro with the specified number of arguments, the paramater list of the nested function instance is parsed, and `#2` is defined to be a macro expanding to that instance. In this case the specified template is responsible for picking up the requested number of arguments. (This is *not* checked.)

```
\cs_new_nopar:Npn \TP_templdecl_setup_f_key:Nn#1#2{
```

`##1` can either be arbitrary inline code, in which case it will be defined with something similar to `\newcommand[val]` so it needs to use `#1` – `#val`.

define it locally here (why this, David???)

```
  \TP_templdecl_define_key:n
  { \TP_templdecl_define_function:NNn#1#2{##1} }
}
```

`P_templdecl_define_function:NNn` Use number of arguments to define function.

```
\cs_new_nopar:Npn \TP_templdecl_define_function:NNn#1#2#3{
  \cs_generate_from_arg_count:NNnn #2 \cs_set:Npn {#1}{#3}
  \toks_put_right:Nf \l_TP_KV_assignments_toks  {
    \cs_generate_from_arg_count:NNnn #2 \cs_set:Npn {#1}{#3}
  }
}
```

`\TP_templdecl_setup_n_key:N` Here is the extended version that tries to deal with `\MultiSelection`.

In case of 'n' keys there is no evaluation at declaration time so it is not sensible to look for `\DelayEvaluation`. For this reason as well as for the fact that `\TP_multiselection_add:nnnnnn` above assumes that it deals with registers that can be accessed via `\toks_use:N` we have to use a different command to handle the `\MultiSelection` args but it is essentially doing the same.

```
\cs_new_nopar:Npn \TP_templdecl_setup_n_key:N#1{
  \TP_templdecl_define_key:n{
    \tl_if_head_eq_meaning:nNTF{##1..}\MultiSelection
    {
      \group_begin:
        \TP_templdecl_multiselection:nnnn ##1
      \group_end:
```

Extracting the correct item from the `\if_case:w` we are building requires a bit of care. Firstly we want to expand the appropirate number of times to get to the item but we also want to ensure we do not have any unwanted leftover `\fi:`s or other junk which is bound to cause errors later on. Therefore we start an `f` expansion (so we don't have to count `\exp_after:wN`s and then stop it again when we want to.

```
      \toks_put_right:Nx\l_TP_KV_assignments_toks {
        \exp_not:n{\tl_set:Nf #1} { \toks_use:N \g_TP_multiselection_toks}
      }
    }
    {
      \cs_set_nopar:Npn #1{##1}                      % setting it?
      \toks_put_right:Nn \l_TP_KV_assignments_toks
                         { \tl_set:Nn #1{##1} }
    }
  }
}
```

`P_templdecl_multiselection:nnnn` Start the `\if_case:w`. When the item is retrieved using an `f` type expansion we better stop it at the right place once we have emerged on the other side of the conditional.

```
\cs_new_nopar:Npn \TP_templdecl_multiselection:nnnn #1#2#3#4{
  \toks_gset:Nn \g_TP_multiselection_toks {\if_case:w #2}
  \clist_map_inline:nn {#3}{
```

```
    \TP_multiselection_add_or_case:n {\exp_stop_f:##1}
  }
  \toks_gput_right:Nn\g_TP_multiselection_toks {
    \else: \use_i_after_fi:nw { \exp_stop_f: #4} \fi:
  }
}
```

\DeclareInstance {type}{instname}{templatename}{keyval}

```
\cs_new_nopar:Npn \DeclareInstance { \DeclareCollectionInstance{} }
```

\DeclareCollectionInstance {collection}{type}{instname}{templatename}{keyval} The fifth argument is picked up implicitly.

```
\cs_new:Npn \DeclareCollectionInstance#1#2#3#4{
  \TP_declare_instance:cnn { <#1>#2/#3 }{ #2/#4 }
}
```

\UseCollection {type}{collection}

```
\cs_new_nopar:Npn \UseCollection#1#2{
  \tl_set:cx { TP@<#1> }
    { {#2} \TP_get_arg_count:n{#1} }
}
```

\TP_let_instance:Nnn \internalcommand{type}{instname}

The way this macro is used, it must result in \cs_set_eq:NwN <csname1> <csname2> after exactly two expansions as it is used this way in \TP_templdecl_eval_i_key_value:nnn!

```
\cs_new_nopar:Npn \TP_let_instance:Nnn#1#2#3{
  \cs_set_eq:Nc #1
  {
    \cs_if_free:cTF { \TP_get_csname_prefix:n{#2} #3 }
      { <>#2/ }
      { \TP_get_csname_prefix:n{#2} }
    #3
  }
}
```

\UseInstance {type}{instname}

```
\cs_new_nopar:Npn \UseInstance#1#2{
  \TP_let_instance:Nnn \l_tmpa_tl {#1}{#2}
  \tl_if_eq:NNTF \l_tmpa_tl \scan_stop:
     \INSTANCEundefined
     \l_tmpa_tl
}
```

ldecl_declare_tmp_instance:nnnn This macro is called when we have seen a `\UseTemplate` declaration as part of an `i` key value. Therefore the first argument will be dropped (it contains the token `\UseTemplate`) the second and third will be combined to refer to the template and the fourth argument will be implictly picked up by `\TP_declare_instance:Nnn`.

```
\cs_new:Npn \TP_templdecl_declare_tmp_instance:nnnn#1#2#3{%
  \TP_declare_instance:Nnn \g_tmpa_tl {#2/#3} }
```

\ShowTemplate Some extension to `\ShowTemplate` so that we also get to see the restrictions if any

```
\cs_new_nopar:Npn \ShowTemplate#1#2{
   \iow_term:x{*******~ Template:~ #1/#2~ *******}
   \iow_term:x{*}
   \iow_term:x{*~ Defaults:}
   \iow_term:x{*}
   \iow_term:x{\token_to_str:N\TPD>/#1/#2=
      \cs_meaning:c{TPD>/#1/#2}}
   \iow_term:x{*}
   \iow_term:x{*~ Restrictions:}
   \iow_term:x{*}
   \iow_term:x{\token_to_str:N\TPR>/#1/#2=
      \cs_meaning:c{TPR>/#1/#2}}
   \iow_term:x{*}
   \iow_term:x{*~ Body:}
   \iow_term:x{*}
   \cs_show:c {TP>/#1/#2}}
```

\ShowCollectionInstance

```
\cs_new_nopar:Npn \ShowCollectionInstance#1#2#3{
   \iow_term:x{*******~ Instance:~ <#1>#2/#3~ *******}
   \iow_term:x{*}
   \cs_show:c {<#1>#2/#3}}
\cs_new_nopar:Npn \ShowInstance{\ShowCollectionInstance{}}
```

decl_setup_fakeregister_key:NNn {setcomand}{privateregister}{key}{internalcode}

```
\cs_new_nopar:Npn \TP_templdecl_setup_fakeregister_key:NNn#1#2#3{
  \TP_templdecl_define_key:n{
    \tl_if_head_eq_meaning:nNTF{##1..}\DelayEvaluation
    {
```

In the v0.08 version of `template.dtx` a `\DelayEvaluation` for a faked register would simply be equiv to a `\cs_set_nopar:Npn` (code is below commented out). The negative side effect of this is that something like `=L` used with `\DelayEvaluation` would not allow for calc syntax since it would end up as `\cs_set_nopar:Npn \foo{a+b}`. The code below changes this to first assign to a scratch register (at runtime) and then do an `\edef`. Could be coded differently to save space (at cost of time)

```
%       \toks_put_right:Nn \l_TP_KV_assignments_toks {\cs_set_nopar:Npn #3{##1}}
%      \toks_put_right:Nn \l_TP_KV_assignments_toks
%                          {#1#2{##1}\cs_set_nopar:Npx #3{\toks_use:N#2}}
      \toks_set:No \l_tmpa_toks {\use_ii:nn ##1}
      \toks_put_right:Nx \l_TP_KV_assignments_toks {
        \exp_not:n{#1#2}{\toks_use:N \l_tmpa_toks}
          \exp_not:n{ \cs_set_nopar:Npx #3{\toks_use:N#2} }

      }
    }
```

Otherwise same game for fake registers except that instead of passing the register to `\TP_multiselection_add:nnnnnn` we pass a temp fake one and doing a def instead of using `\setlength` or `\setcounter`

and i haven't done the `\DelayEvaluation` bit for that case! as i'm not sure what the best approach is for those things[9]

```
    {
      \tl_if_head_eq_meaning:nNTF{##1..}\MultiSelection
      {
        \group_begin:
          \TP_multiselection_add:nnnnnn#1#2##1
        \group_end:
        \toks_put_right:Nx\l_TP_KV_assignments_toks
          {\exp_not:n{\cs_set_nopar:Npn #3} {\toks_use:N\g_TP_multiselection_toks}}
      }
      {
        #1#2{##1}
        \toks_put_right:Nx\l_TP_KV_assignments_toks
                    {\exp_not:n{\cs_set_nopar:Npn#3} {\toks_use:N#2}}
      }
    }
  }
}
```

`\g_TP_multiselection_toks`

```
\toks_new:N \g_TP_multiselection_toks
```

`\TP_multiselection_add:nnnnnn` {⟨*operation*⟩} {⟨*register*⟩} `\MultiSelection` {⟨*selector*⟩} {⟨*case-list*⟩} {⟨*else-case*⟩}

This command builds up the `\if_case:w` code from the three arguments of `\MultiSelection` and stores it in `\g_TP_multiselection_toks`. This code is supposed to be run in a group so a) we don't have to initialise `\g_TP_multiselection_toks` and b) all changes to the used registers not affecting the outside.

```
\cs_new_nopar:Npn \TP_multiselection_add:nnnnnn #1#2#3#4#5#6{
```

[9] we might disallow it for that case in general — not a nice rule but an explainable one

```
  \toks_gset:Nn \g_TP_multiselection_toks {\if_case:w #4}
  \clist_map_inline:nn {#5}{
    \tl_if_head_eq_meaning:nNTF{##1..}\DelayEvaluation
    {
      \TP_multiselection_add_or_case:n {##1}
    }
    {
      #1#2{##1}
      \TP_multiselection_add_or_case:o { \toks_use:N #2 }
    }
  }
  \toks_gput_right:Nn \g_TP_multiselection_toks {
    \else: \use_i_after_fi:nw{#6}\fi:
  }
}
```

`TP_multiselection_add_or_case:o` No need to worry about where `\or:` is allowed to be added since all loops in LaTeX3 process the item outside conditionals.

```
\cs_new_nopar:Npn \TP_multiselection_add_or_case:n #1 {
  \toks_gput_right:Nn \g_TP_multiselection_toks {
    \or: \use_i_after_orelse:nw{#1}
  }
}
\cs_generate_variant:Nn \TP_multiselection_add_or_case:n {o}
```

Since i like to set things like `item-label-text` using this mechanism i need to handle the 'n' key specially.

Actually i could have probably extended `\TP_templdecl_setup_f_key:nnN` thus making this generally available to all f⟨*number*⟩ keys but was too lazy (or too stupid) to get it right first time so settled for the simple solution.

So `\TP_templdecl_parse_KV:nnw` now calls `\TP_templdecl_setup_n_key:nN` for the 'n' key. looks like this thus be fixed some time soon

`\IfExistsInstanceTF` tests that there is a *default* definition taken from xinitials.dtx:

```
\cs_new_nopar:Npn \IfExistsInstanceTF#1#2{
  \cs_if_exist:cTF{<>#1/#2}
}
```

FMi: what happens if we are in collection FOO and there exists an instance I for type T within this collection but there doesn't exist an instance in the empty collection?

What would happen if ... — not clear to me what the sematics really should be. The code below is not better only different(and slower).[10]

[10] fix semantics

```
\cs_set_nopar:Npn \IfExistsInstanceTF#1#2{
  \TP_let_instance:Nnn \l_tmpa_tl {#1}{#2}
% next is not \tl_if_eq:NNTF but ...FT so done manually
  \if_meaning:w\l_tmpa_tl\scan_stop:
    \exp_after:wN\use_ii:nn
  \else:
    \exp_after:wN\use_i:nn
  \fi:}
```

\DeclareRestrictedTemplate

Setting it up:

```
 \DeclareRestrictedTemplate
{T-type} {new-T-name} {source-T-name} {keyvals}
```

This uses the same code as T-type source-T-name but adds settings from keyvals

```
\cs_new_nopar:Npn \DeclareRestrictedTemplate#1#2#3#4{
  % CCC do we need a group here??
  \tl_set_eq:cc { TPD>/#1/#2 } { TPD>/#1/#3 }
  \tl_set_eq:cc {  TP>/#1/#2 } {  TP>/#1/#3 }
  \toks_set:Nv \l_TP_KV_assignments_toks { TPR>/ #1 / #3 }
% adds stuff to \l_TP_KV_assignments_toks
  \setkeys {\cs:w TPO>/#1/#3\cs_end:}{#4}

  \tl_set:cx { TPR>/#1/#2 }
              { \toks_use:N \l_TP_KV_assignments_toks }
  \cs_if_free:cTF { TPO>/#1/#3 }
    { \tl_set:cn {TPO>/#1/#2}{#1/#3}       }
    { \tl_set_eq:cc {TPO>/#1/#2}{TPO>/#1/#3}  }
}
```

Internals:

\TP_instdecl_generate_assignments:nn

These could probably be inlined, even when they do something!

Assumption: setkeys fully expands its first argument.

```
\cs_new_nopar:Npn \TP_instdecl_generate_assignments:nn#1#2 {
                                % Returns to \l_TP_KV_assignments_toks
                                % the restrictions
                                % stored in the TP-structure (at present
                                % in YAM) of the template #1
    \exp_args:NNo \toks_set:No \l_TP_default_assignments_toks
                  {\cs:w TPD>/#1\cs_end:\scan_stop:\scan_stop:}
    \toks_set:Nv \l_TP_KV_assignments_toks { TPR>/ #1 }
    \setkeys { \cs:w TPO>/#1 \cs_end: }
             { #2 }             % adds stuff to \l_TP_KV_assignments_toks

% prepends stuff to \l_TP_KV_assignments_toks :
    \exp_after:wN\TP_instdecl_add_default_recurse:nn
```

```
        \toks_use:N\l_TP_default_assignments_toks

}
```

`instdecl_add_default_recurse:nn` [2001/06/10 Think about doing this properly with explicit plists! — but this means that one has to think about whether or not plists should be implemented as token registers and not as tl vars as they are now.]

```
\cs_new_nopar:Npn \TP_instdecl_add_default_recurse:nn#1#2{
  \token_if_eq_meaning:NNF #1\scan_stop:
  {
    \l_tmpa_toks{#2}
    \tl_set:Nx \l_tmpa_tl {
      {\toks_use:N \l_tmpa_toks \toks_use:N \l_TP_KV_assignments_toks}
    }
    \l_TP_KV_assignments_toks \l_tmpa_tl
    \TP_instdecl_add_default_recurse:nn
  }
}
```

`TPD>/type/name` stores the default parameter assignments.

`TPR>/type/name` stores the parameter assignments that have been made for a restricted template otherwise it is undefined (or `\scan_stop:`).

`TPO>/type/name` stores the full name (i.e. as `type/name`) of the template a restricted template is coming from originally.

`TP_split_finite_skip_value:nnNN` This macro is for use in error checking template values like `text-float-sep` that can't contain infinite glue and needs the shrink and/or stretch components. First argument is the skip register (which is likely to be user input), second is a template key name, and the last two are the ⟨*dimen*⟩ registers that stores the stretch and shrink components. Assignments are global.

```
\cs_new_nopar:Npn \TP_split_finite_skip_value:nnNN #1#2{
  \skip_split_finite_else_action:nnNN {#1} {
    \PackageError{template}{Value~ for~ key~ #2~ contains~ ‘fil(ll)’}
    {Only~ finite~ minus~ or~ plus~ parts~ are~ allowed~ for~ this~ key.}
  }
}
```

⟨/package⟩