

The `ltemplates.dtx` code*

Frank Mittelbach, Chris Rowley, David Carlisle, L^AT_EX Project[†]

October 31, 2024

1 Introduction

There are three broad “layers” between putting down ideas into a source file and ending up with a typeset document. These layers of document writing are

1. authoring of the text with mark-up;
2. document layout design;
3. implementation (with T_EX programming) of the design.

We write the text as an author, and we see the visual output of the design after the document is generated; the T_EX implementation in the middle is the glue between the two.

L^AT_EX’s greatest success has been to standardise a system of mark-up that balances the trade-off between ease of reading and ease of writing to suit almost all forms of technical writing. It’s other original strength was a good background in typographical design; while the standard L^AT_EX 2_ε classes look somewhat dated now in terms of their visual design, their typography is generally sound (barring the occasional minor faults).

However, L^AT_EX 2_ε has always lacked a standard approach to customising the visual design of a document. Changing the looks of the standard classes involved either:

- Creating a new version of the implementation code of the class and editing it.
- Loading one of the many packages to customise certain elements of the standard classes.
- Loading a completely different document class, such as KOMA-Script or memoir, that allows easy customisation.

All three of these approaches have their drawbacks and learning curves.

The idea behind `ltemplates` is to cleanly separate the three layers introduced at the beginning of this section, so that document authors who are not programmers can easily change the design of their documents. `ltemplates` also makes it easier for L^AT_EX programmers to provide their own customisations on top of a pre-existing class.

*This file has version v1.0d dated 2024-10-07, © L^AT_EX Project.

†E-mail: latex-team@latex-project.org

2 What is a document?

Besides the textual content of the words themselves, the source file of a document contains mark-up elements that add structure to the document. These elements include sectional divisions, figure/table captions, lists of various sorts, theorems/proofs, and so on. The list will be different for every document that can be written.

Each element can be represented logically without worrying about the formatting, with mark-up such as `\section`, `\caption`, `\begin{enumerate}` and so on. The output of each one of these document elements will be a typeset representation of the information marked up, and the visual arrangement and design of these elements can vary widely in producing a variety of desired outcomes.

For each type of document element, there may be design variations that contain the same sort of information but present it in slightly different ways. For example, the difference between a numbered and an unnumbered section, `\section` and `\section*`, or the difference between an itemised list or an enumerated list.

There are three distinct layers in the definition of “a document” at this level

1. semantic elements such as the ideas of sections and lists;
2. a set of design solutions for representing these elements visually;
3. specific variations for these designs that represent the elements in the document.

In the parlance of the template system, these are called types, templates, and instances, and they are discussed below in sections 4, 5, and 7, respectively.

3 Types, templates, and instances

By formally declaring documents to be composed of mark-up elements grouped into types, which are interpreted and typeset with a set of templates, each of which has one or more instances with which to compose each and every semantic unit of the text, we can cleanly separate the components of document construction.

All of the structures provided by the template system are global, and do not respect \TeX grouping.

4 Template types

An *template type* (sometimes just “type”) is an abstract idea of a document element that takes a fixed number of arguments corresponding to the information from the document author that it is representing. A sectioning type, for example, might take three inputs: “title”, “short title”, and “label”.

Any given document class will define which types are to be used in the document, and any template of a given type can be used to generate an instance for the type. (Of course, different templates will produce different typeset representations, but the underlying content will be the same.)

`\NewTemplateType` `\NewTemplateType {<template type>} {<no. of args>}`

This function defines an *<template type>* taking *<number of arguments>*, where the *<type>* is an abstraction as discussed above. For example,

```
\NewTemplateType{sectioning}{3}
```

creates a type “sectioning”, where each use of that type will need three arguments.

5 Templates

A *template* is a generalised design solution for representing the information of a specified type. Templates that do the same thing, but in different ways, are grouped together by their type and given separate names. There are two important parts to a template:

- the parameters it takes to vary the design it is producing;
- the implementation of the design.

As a document author or designer does not care about the implementation but rather only the interface to the template, these two aspects of the template definition are split into two independent declarations, `\DeclareTemplateInterface` and `\DeclareTemplateCode`.

`\DeclareTemplateInterface` `\DeclareTemplateInterface`
`{<type>} {<template>} {<no. of args>}`
`{<key list>}`

A *<template>* interface is declared for a particular *<type>*, where the *<number of arguments>* must agree with the type declaration. The interface itself is defined by the *<key list>*, which is itself a key–value list taking a specialized format:

```
<key1> : <key type1> ,  
<key2> : <key type2> ,  
<key3> : <key type3> = <default3> ,  
<key4> : <key type4> = <default4> ,  
...
```

Each *<key>* name should consist of ASCII characters, with the exception of `,`, `=` and `␣`. The recommended form for key names is to use lower case letters, with dashes to separate out different parts. Spaces are ignored in key names, so they can be included or missed out at will. Each *<key>* must have a *<key type>*, which defined the type of input that the *<key>* requires. A full list of key types is given in Table 1. Each key may have a *<default>* value, which will be used in by the template if the *<key>* is not set explicitly. The *<default>* should be of the correct form to be accepted by the *<key type>* of the *<key>*: this is not checked by the code. Expressions for numerical values are evaluated when the template is used, thus for example values given in terms of `em` or `ex` will be set respecting the prevailing font.

Key-type	Description of input
<code>boolean</code>	<code>true</code> or <code>false</code>
<code>choice{⟨choices⟩}</code>	A list of pre-defined <code>⟨choices⟩</code>
<code>commalist</code>	A comma-separated list
<code>function{⟨N⟩}</code>	A function definition with N arguments (N from 0 to 9)
<code>instance{⟨name⟩}</code>	An instance of type <code>⟨name⟩</code>
<code>integer</code>	An integer or integer expression
<code>length</code>	A fixed length
<code>muskip</code>	A math length with shrink and stretch components
<code>real</code>	A real (floating point) value
<code>skip</code>	A length with shrink and stretch components
<code>tokenlist</code>	A token list: any text or commands

Table 1: Key-types for defining template interfaces with `\DeclareTemplateInterface`.

`\KeyValue` `\KeyValue {⟨key name⟩}`

There are occasions where the default (or value) for one key should be taken from another. The `\KeyValue` function can be used to transfer this information without needing to know the internal implementation of the key:

```

\DeclareTemplateInterface { type } { template } { no. of args }
{
  key-name-1 : key-type = value ,
  key-name-2 : key-type = \KeyValue { key-name-1 },
  ...
}

```

Key-type	Description of binding
	Boolean variable, <i>e.g.</i> <code>\l_tmpa_bool</code>
<code>choice</code>	List of choice implementations (see Section 6)
<code>commalist</code>	Comma list, <i>e.g.</i> <code>\l_tmpa_clist</code>
<code>function</code>	Function taking N arguments, <i>e.g.</i> <code>\use_i:nn</code>
<code>instance</code>	
<code>integer</code>	Integer variable, <i>e.g.</i> <code>\l_tmpa_int</code>
<code>length</code>	Dimension variable, <i>e.g.</i> <code>\l_tmpa_dim</code>
<code>muskip</code>	Muskip variable, <i>e.g.</i> <code>\l_tmpa_muskip</code>
<code>real</code>	Floating-point variable, <i>e.g.</i> <code>\l_tmpa_fp</code>
<code>skip</code>	Skip variable, <i>e.g.</i> <code>\l_tmpa_skip</code>
<code>tokenlist</code>	Token list variable, <i>e.g.</i> <code>\l_tmpa_tl</code>

Table 2: Bindings required for different key types when defining template implementations with `\DeclareTemplateCode`. Apart from `code`, `choice` and `function` all of these accept the key word `global` to carry out a global assignment.

```

\DeclareTemplateCode \DeclareTemplateCode
  <{type}> <{template}> <{no. of args}>
  <{key bindings}> <{code}>

```

The relationship between a templates keys and the internal implementation is created using the `\DeclareTemplateCode` function. As with `\DeclareTemplateInterface`, the `<template>` name is given along with the `<type>` and `<number of arguments>` required. The `<key bindings>` argument is a key–value list which specifies the relationship between each `<key>` of the template interface with an underlying `<variable>`.

```

  <key1> = <variable1>,
  <key2> = <variable2>,
  <key3> = global <variable3>,
  <key4> = global <variable4>,
  ...

```

With the exception of the `choice`, `code` and `function` key types, the `<variable>` here should be the name of an existing L^AT_EX3 register. As illustrated, the key word “global” may be included in the listing to indicate that the `<variable>` should be assigned globally. A full list of variable bindings is given in Table 2.

The `<code>` argument of `\DeclareTemplateCode` is used as the replacement text for the template when it is used, either directly or as an instance. This may therefore accept arguments `#1`, `#2`, *etc.* as detailed by the `<number of arguments>` taken by the type.

```

\AssignTemplateKeys \AssignTemplateKeys

```

In the final argument of `\DeclareTemplateCode` the assignment of keys defined by the template may be delayed by including the command `\AssignTemplateKeys`. If this is *not* present, keys are assigned immediately before the template code. If `\AssignTemplateKeys` is present, assignment is delayed until this point. Note that the command must be *directly* present in the code, not placed within a nested command/macro.

```
\DeclareTemplateCopy \DeclareTemplateCopy
  {<type>} {<template2>} {<template1>}
```

Copies *<template1>* of *<type>* to a new name *<template2>*: the copy can then be edited independent of the original.

6 Multiple choices

The `choice` key type implements multiple choice input. At the interface level, only the list of valid choices is needed:

```
\DeclareTemplateInterface { foo } { bar } { 0 }
  { key-name : choice { A, B, C } }
```

where the choices are given as a comma-list (which must therefore be wrapped in braces). A default value can also be given:

```
\DeclareTemplateInterface { foo } { bar } { 0 }
  { key-name : choice { A, B, C } = A }
```

At the implementation level, each choice is associated with code, using a nested key-value list.

```
\DeclareTemplateCode { foo } { bar } { 0 }
  {
    key-name =
      {
        A = Code-A ,
        B = Code-B ,
        C = Code-C
      }
  }
  { ... }
```

The two choice lists should match, but in the implementation a special `unknown` choice is also available. This can be used to ignore values and implement an “else” branch:

```
\DeclareTemplateCode { foo } { bar } { 0 }
  {
    key-name =
      {
        A      = Code-A ,
        B      = Code-B ,
        C      = Code-C ,
        unknown = Else-code
      }
  }
  { ... }
```

The `unknown` entry must be the last one given, and should *not* be listed in the interface part of the template.

For keys which accept the values `true` and `false` both the boolean and choice key types can be used. As template interfaces are intended to prompt clarity at the design level, the boolean key type should be favoured, with the choice type reserved for keys which take arbitrary values.

7 Instances

After a template is defined it still needs to be put to use. The parameters that it expects need to be defined before it can be used in a document. Every time a template has parameters given to it, an *instance* is created, and this is the code that ends up in the document to perform the typesetting of whatever pieces of information are input into it.

For example, a template might say “here is a section with or without a number that might be centred or left aligned and print its contents in a certain font of a certain size, with a bit of a gap before and after it” whereas an instance declares “this is a section with a number, which is centred and set in 12 pt italic with a 10 pt skip before and a 12 pt skip after it”. Therefore, an instance is just a frozen version of a template with specific settings as chosen by the designer.

```
\DeclareInstance \DeclareInstance
  {\type} {\instance} {\template} {\parameters}
```

This function uses a $\langle\text{template}\rangle$ for an $\langle\text{type}\rangle$ to create an $\langle\text{instance}\rangle$. The $\langle\text{instance}\rangle$ will be set up using the $\langle\text{parameters}\rangle$, which will set some of the $\langle\text{keys}\rangle$ in the $\langle\text{template}\rangle$.

As a practical example, consider a type for document sections (which might include chapters, parts, sections, *etc.*), which is called `sectioning`. One possible template for this type might be called `basic`, and one instance of this template would be a numbered section. The instance declaration might read:

```
\DeclareInstance { sectioning } { section-num } { basic }
{
  numbered      = true ,
  justification = center ,
  font          =\normalsize\itshape ,
  before-skip  = 10pt ,
  after-skip   = 12pt ,
}
```

Of course, the key names here are entirely imaginary, but illustrate the general idea of fixing some settings.

```
\IfInstanceExistsT \IfInstanceExistsTF {\type} {\instance} {\true code} {\false code}
\IfInstanceExistsF
\IfInstanceExistsTF Tests if the named  $\langle\text{instance}\rangle$  of a  $\langle\text{type}\rangle$  exists, and then inserts the appropriate code
into the input stream.
```

```
\DeclareInstanceCopy \DeclareInstanceCopy
  {\type} {\instance2} {\instance1}
Copies the  $\langle\text{values}\rangle$  for  $\langle\text{instance1}\rangle$  for an  $\langle\text{type}\rangle$  to  $\langle\text{instance2}\rangle$ .
```

8 Document interface

After the instances have been chosen, document commands must be declared to use those instances in the document. `\UseInstance` calls instances directly, and this command should be used internally in document-level mark-up.

\UseInstance `\UseInstance`
`{<type>} {<instance>} <arguments>`

Uses an `<instance>` of the `<type>`, which will require `<arguments>` as determined by the number specified for the `<type>`. The `<instance>` must have been declared before it can be used, otherwise an error is raised.

\UseTemplate `\UseTemplate {<type>} {<template>}`
`{<settings>} <arguments>`

Uses the `<template>` of the specified `<type>`, applying the `<settings>` and absorbing `<arguments>` as detailed by the `<type>` declaration. This in effect is the same as creating an instance using `\DeclareInstance` and immediately using it with `\UseInstance`, but without the instance having any further existence. It is therefore useful where a template needs to be used once.

This function can also be used as the argument to `instance` key types:

```
\DeclareInstance { type } { template } { instance }
{
  instance-key =
    \UseTemplate { type2 } { template2 } { <settings> }
}
```

9 Changing existing definitions

Template parameters may be assigned specific defaults for instances to use if the instance declaration doesn't explicit set those parameters. In some cases, the document designer will wish to edit these defaults to allow them to “cascade” to the instances. The alternative would be to set each parameter identically for each instance declaration, a tedious and error-prone process.

\EditTemplateDefaults `\EditTemplateDefaults`
`{<type>} {<template>} {<new defaults>}`

Edits the `<defaults>` for a `<template>` for an `<type>`. The `<new defaults>`, given as a key–value list, replace the existing defaults for the `<template>`. This means that the change will apply to instances declared after the editing, but that instances which have already been created are unaffected.

\EditInstance `\EditInstance`
`{<type>} {<instance>} {<new values>}`

Edits the `<values>` for an `<instance>` for an `<type>`. The `<new values>`, given as a key–value list, replace the existing values for the `<instance>`. This function is complementary to `\EditTemplateDefaults`: `\EditInstance` changes a single instance while leaving the template untouched.

10 *Ad hoc* adjustment of templates

`\SetTemplateKeys` `\SetTemplateKeys` $\langle type \rangle$ $\langle template \rangle$ $\langle keyvals \rangle$

At point of use it may be useful to apply changed to individual instances. This is supported as each template key is made available for adjustment using `\SetTemplateKeys`.

For example, after

```
\NewTypeType{MyObj}{0}
\DeclareTemplateInterface{MyObj}{TemplateA}{0}
{
  akey: tokenlist ,
  bkey: function{2}
}
\DeclareTemplateCode{MyObj}{TemplateA}{0}
{
  akey = SomeTokens ,
  bkey = \func:nn ,
}
```

the template keys could be adjusted in an *ad hoc* fashion using

```
\SetTemplateKeys{MyObj}{TemplateA}
{
  akey = OtherTokens ,
  bkey = \AltFunc:nn
}
```

11 Getting information about templates and instances

`\ShowInstanceValues` `\ShowInstanceValues` $\langle type \rangle$ $\langle instance \rangle$

Shows the $\langle values \rangle$ for an $\langle instance \rangle$ of the given $\langle type \rangle$ at the terminal.

`\ShowTemplateCode` `\ShowTemplateCode` $\langle type \rangle$ $\langle template \rangle$

Shows the $\langle code \rangle$ of a $\langle template \rangle$ for an $\langle type \rangle$ in the terminal.

`\ShowTemplateDefaults` `\ShowTemplateDefaults` $\langle type \rangle$ $\langle template \rangle$

Shows the $\langle default \rangle$ values of a $\langle template \rangle$ for an $\langle type \rangle$ in the terminal.

`\ShowTemplateInterface` `\ShowTemplateInterface` $\langle type \rangle$ $\langle template \rangle$

Shows the $\langle keys \rangle$ and associated $\langle key types \rangle$ of a $\langle template \rangle$ for an $\langle type \rangle$ in the terminal.

`\ShowTemplateVariables` `\ShowTemplateVariables {<type>} {<template>}`

Shows the `<variables>` and associated `<keys>` of a `<template>` for an `<type>` in the terminal. Note that `code` and `choice` keys do not map directly to variables but to arbitrary code. For `choice` keys, each valid choice is shown as a separate entry in the list, with the key name and choice separated by a space, for example

```
Template 'example' of type 'example' has variable mapping:
> demo unknown => \def \demo {?}
> demo c => \def \demo {c}
> demo b => \def \demo {b}
> demo a => \def \demo {a}.
```

would be shown for a choice key `demo` with valid choices `a`, `b` and `c`, plus code for an `unknown` branch.

12 The implementation

```
1 <@@=template>
2 <*2kernel>
3 \message{templates,}
4 </2kernel>
5 <*2kernel | latexrelease>
6 \ExplSyntaxOn
7 <latexrelease>\NewModuleRelease{2024/06/01}{lttemplates}
8 <latexrelease> {Prototype-document-commands}%
```

12.1 Variables and constants

```
\c__template_code_root_tl
\c__template_defaults_root_tl
\c__template_instances_root_tl
\c__template_keytypes_root_tl
\c__template_key_order_root_tl
\c__template_restrict_root_tl
\c__template_values_root_tl
\c__template_vars_root_tl
```

So that literal values are kept to a minimum.

```
9 \tl_const:Nn \c__template_code_root_tl { template~code~>~ }
10 \tl_const:Nn \c__template_defaults_root_tl { template~defaults~>~ }
11 \tl_const:Nn \c__template_instances_root_tl { template~instance~>~ }
12 \tl_const:Nn \c__template_keytypes_root_tl { template~key~types~>~ }
13 \tl_const:Nn \c__template_key_order_root_tl { template~key~order~>~ }
14 \tl_const:Nn \c__template_values_root_tl { template~values~>~ }
15 \tl_const:Nn \c__template_vars_root_tl { template~vars~>~ }
```

\c__template_keytypes_arg_seq

A list of keytypes which also need additional data (an argument), used to parse the keytype correctly.

```
16 \seq_const_from_clist:Nn \c__template_keytypes_arg_seq
17 { choice , function , instance }
```

\g__template_type_prop

For storing types and the associated number of arguments.

```
18 \prop_new:N \g__template_type_prop
```

\l__template_assignments_tl

When creating an instance, the assigned values are collected here.

```
19 \tl_new:N \l__template_assignments_tl
```

\l__template_default_tl

The default value for a key is recovered here from the property list in which it is stored.

```
20 \tl_new:N \l__template_default_tl
```

\l__template_error_bool

A flag for errors to be carried forward.

```
21 \bool_new:N \l__template_error_bool
```

\l__template_global_bool

Used to indicate that assignments should be global.

```
22 \bool_new:N \l__template_global_bool
```

\l__template_key_name_tl

\l__template_keytype_tl

\l__template_keytype_arg_tl

\l__template_value_tl

\l__template_var_tl

When defining each key in a template, the name and type of the key need to be separated and stored. Any argument needed by the keytype is also stored separately.

```
23 \tl_new:N \l__template_key_name_tl
24 \tl_new:N \l__template_keytype_tl
25 \tl_new:N \l__template_keytype_arg_tl
26 \tl_new:N \l__template_value_tl
27 \tl_new:N \l__template_var_tl
```

\l__template_keytypes_prop

\l__template_key_order_seq

\l__template_values_prop

\l__template_vars_prop

To avoid needing too many difficult-to-follow csname assignments, various scratch token registers are used to build up data, which is then transferred

```
28 \prop_new:N \l__template_keytypes_prop
29 \seq_new:N \l__template_key_order_seq
30 \prop_new:N \l__template_values_prop
31 \prop_new:N \l__template_vars_prop
```

```

\l__template_tmp_clist Scratch space.
\l__template_tmp_dim      32 \clist_new:N \l__template_tmp_clist
\l__template_tmp_int      33 \dim_new:N \l__template_tmp_dim
\l__template_tmp_muskip   34 \int_new:N \l__template_tmp_int
\l__template_tmp_skip     35 \muskip_new:N \l__template_tmp_muskip
\l__template_tmp_tl       36 \skip_new:N \l__template_tmp_skip
                           37 \tl_new:N \l__template_tmp_tl

```

```

\s__template_mark Internal scan marks.
\s__template_stop   38 \scan_new:N \s__template_mark
                   39 \scan_new:N \s__template_stop

```

```

\q__template_nil Internal quarks.
                 40 \quark_new:N \q__template_nil

```

```

\__template_quark_if_nil_p:n Branching quark conditional.
\__template_quark_if_nil:nTF  41 \__kernel_quark_new_conditional:Nn \__template_quark_if_nil:N { F }

(End of definition for \__template_quark_if_nil:nTF.)

```

12.2 Testing existence and validity

There are a number of checks needed for either the existence of a type, template or instance. There are also some for the validity of a particular call. All of these are collected up here.

```

\__template_execute_if_arg_agree:nnT A test agreement between the number of arguments for the template type and that specified when creating a template. This is not done as a separate conditional for efficiency and better error message
                                     42 \cs_new_protected:Npn \__template_execute_if_arg_agree:nnT #1#2#3
                                     43 {
                                     44   \prop_get:NnN \g__template_type_prop {#1} \l__template_tmp_tl
                                     45   \int_compare:nNnTF {#2} = \l__template_tmp_tl
                                     46     {#3}
                                     47     {
                                     48       \msg_error:nneee { template } { argument-number-mismatch }
                                     49       {#1} { \l__template_tmp_tl } {#2}
                                     50     }
                                     51   }

```

(End of definition for __template_execute_if_arg_agree:nnT.)

```

\__template_execute_if_code_exist:nnT A template is only fully declared if the code has been set up, which can be checked by looking for the template function itself.
                                       52 \cs_new_protected:Npn \__template_execute_if_code_exist:nnT #1#2#3
                                       53 {
                                       54   \cs_if_exist:cTF { \c__template_code_root_tl #1 / #2 }
                                       55   {#3}

```

```

56     { \msg_error:nnnn { template } { no-template-code } {#1} {#2} }
57   }

```

(End of definition for `__template_execute_if_code_exist:nnT`.)

`__template_execute_if_keytype_exist:nT` The test for valid keytypes looks for a function to set up the key, which is part of the
`__template_execute_if_keytype_exist:VT` “code” side of the template definition. This avoids having different lists for the two parts
of the process.

```

58 \cs_new_protected:Npn \__template_execute_if_keytype_exist:nT #1#2
59   {
60     \cs_if_exist:cTF { __template_store_value_ #1 :n }
61       {#2}
62     { \msg_error:nnn { template } { unknown-keytype } {#1} }
63   }
64 \cs_generate_variant:Nn \__template_execute_if_keytype_exist:nT { V }

```

(End of definition for `__template_execute_if_keytype_exist:nT`.)

`__template_execute_if_type_exist:nT` To check that a particular type is valid.

```

65 \cs_new_protected:Npn \__template_execute_if_type_exist:nT #1#2
66   {
67     \prop_if_in:NnTF \g__template_type_prop {#1}
68       {#2}
69     { \msg_error:nnn { template } { unknown-type } {#1} }
70   }

```

(End of definition for `__template_execute_if_type_exist:nT`.)

`__template_execute_if_keys_exist:nnT` To check that the keys for a template have been set up before trying to create any code,
a simple check for the correctly-named keytype property list.

```

71 \cs_new_protected:Npn \__template_if_keys_exist:nnT #1#2#3
72   {
73     \cs_if_exist:cTF { \c__template_keytypes_root_tl #1 / #2 }
74       {#3}
75     { \msg_error:nnnn { template } { unknown-template } {#1} {#2} }
76   }

```

(End of definition for `__template_execute_if_keys_exist:nnT`.)

`__template_if_key_value:nTF` Tests for the first token in a string being `\KeyValue`.

`__template_if_key_value:VTF`

```

77 \prg_new_conditional:Npnn \__template_if_key_value:n #1 { T , F , TF }
78   {
79     \str_if_eq:noTF { \KeyValue } { \tl_head:w #1 \q_nil \q_stop }
80     \prg_return_true:
81     \prg_return_false:
82   }
83 \prg_generate_conditional_variant:Nnn \__template_if_key_value:n { V } { T , F , TF }

```

(End of definition for `__template_if_key_value:nTF`.)

`__template_if_instance_exist:nnTF` Testing for an instance

```

84 \prg_new_conditional:Npnn \__template_if_instance_exist:nn #1#2 { T , F , TF }
85   {
86     \cs_if_exist:cTF { \c__template_instances_root_tl #1 / #2 }
87     \prg_return_true:
88     \prg_return_false:
89   }

```

(End of definition for `__template_if_instance_exist:nTF`.)

`__template_if_use_template:nTF` Tests for the first token in a string being `\UseTemplate`.

```
90 \prg_new_conditional:Npnn __template_if_use_template:n #1 { TF }
91 {
92   \str_if_eq:noTF { \UseTemplate } { \tl_head:w #1 \q_nil \q_stop }
93   \prg_return_true:
94   \prg_return_false:
95 }
```

(End of definition for `__template_if_use_template:nTF`.)

12.3 Saving and recovering property lists

The various property lists for templates have to be shuffled in and out of storage.

The defaults and keytypes are transferred from the scratch property lists to the “proper” lists for the template being created.

```
__template_store_defaults:nn
__template_store_keytypes:nn
__template_store_values:nn
__template_store_vars:nn
96 \cs_new_protected:Npn __template_store_defaults:nn #1#2
97 {
98   \debug_suspend:
99   \prop_gclear_new:c { \c__template_defaults_root_tl #1 / #2 }
100   \prop_gset_eq:cN { \c__template_defaults_root_tl #1 / #2 }
101   \l__template_values_prop
102   \debug_resume:
103 }
104 \cs_new_protected:Npn __template_store_keytypes:nn #1#2
105 {
106   \debug_suspend:
107   \prop_if_exist:cTF { \c__template_keytypes_root_tl #1 / #2 }
108   {
109     \msg_info:nnnn { template } { declare-template-interface } {#1} {#2}
110     \prop_gclear:c { \c__template_keytypes_root_tl #1 / #2 }
111   }
112   { \prop_new:c { \c__template_keytypes_root_tl #1 / #2 } }
113   \prop_gset_eq:cN { \c__template_keytypes_root_tl #1 / #2 }
114   \l__template_keytypes_prop
115   \seq_gclear_new:c { \c__template_key_order_root_tl #1 / #2 }
116   \seq_gset_eq:cN { \c__template_key_order_root_tl #1 / #2 }
117   \l__template_key_order_seq
118   \debug_resume:
119 }
120 \cs_new_protected:Npn __template_store_values:nn #1#2
121 {
122   \debug_suspend:
123   \prop_clear_new:c { \c__template_values_root_tl #1 / #2 }
124   \prop_set_eq:cN { \c__template_values_root_tl #1 / #2 }
125   \l__template_values_prop
126   \debug_resume:
127 }
128 \cs_new_protected:Npn __template_store_vars:nn #1#2
129 {
130   \debug_suspend:
131   \prop_gclear_new:c { \c__template_vars_root_tl #1 / #2 }
```

```

132     \prop_gset_eq:cN { \c__template_vars_root_tl #1 / #2 }
133     \l__template_vars_prop
134     \debug_resume:
135 }

```

(End of definition for `__template_store_defaults:nn` and others.)

`__template_recover_defaults:nn` `__template_recover_keytypes:nn` `__template_recover_values:nn` `__template_recover_vars:nn` Recovering the stored data for a template is rather less complex than storing it. All that happens is the data is transferred from the permanent to the scratch storage. However, we need to check the scratch storage does exist.

```

136 \cs_new_protected:Npn \__template_recover_defaults:nn #1#2
137 {
138   \prop_if_exist:cTF
139     { \c__template_defaults_root_tl #1 / #2 }
140     {
141       \prop_set_eq:Nc \l__template_values_prop
142         { \c__template_defaults_root_tl #1 / #2 }
143     }
144     { \prop_clear:N \l__template_values_prop }
145 }
146 \cs_new_protected:Npn \__template_recover_keytypes:nn #1#2
147 {
148   \prop_if_exist:cTF
149     { \c__template_keytypes_root_tl #1 / #2 }
150     {
151       \prop_set_eq:Nc \l__template_keytypes_prop
152         { \c__template_keytypes_root_tl #1 / #2 }
153     }
154     { \prop_clear:N \l__template_keytypes_prop }
155   \seq_if_exist:cTF { \c__template_key_order_root_tl #1 / #2 }
156     {
157       \seq_set_eq:Nc \l__template_key_order_seq
158         { \c__template_key_order_root_tl #1 / #2 }
159     }
160     { \seq_clear:N \l__template_key_order_seq }
161 }
162 \cs_new_protected:Npn \__template_recover_values:nn #1#2
163 {
164   \prop_if_exist:cTF
165     { \c__template_values_root_tl #1 / #2 }
166     {
167       \prop_set_eq:Nc \l__template_values_prop
168         { \c__template_values_root_tl #1 / #2 }
169     }
170     { \prop_clear:N \l__template_values_prop }
171 }
172 \cs_new_protected:Npn \__template_recover_vars:nn #1#2
173 {
174   \prop_if_exist:cTF
175     { \c__template_vars_root_tl #1 / #2 }
176     {
177       \prop_set_eq:Nc \l__template_vars_prop
178         { \c__template_vars_root_tl #1 / #2 }
179     }

```

```

180     { \prop_clear:N \l__template_vars_prop }
181   }

```

(End of definition for __template_recover_defaults:nn and others.)

12.4 Creating new template types

`__template_define_type:nn` Although the type is the “top level” of the template system, it is actually very easy to implement. All that happens is that the number of arguments required is recorded, indexed by the name of the type.

```

182 \cs_new_protected:Npn \__template_define_type:nn #1#2
183   {
184     \prop_if_in:NnTF \g__template_type_prop {#1}
185       { \msg_error:nnn { template } { type-already-defined } {#1} }
186       { \__template_declare_type:nn {#1} {#2} }
187   }
188 \cs_new_protected:Npn \__template_declare_type:nn #1#2
189   {
190     \int_set:Nn \l__template_tmp_int {#2}
191     \int_compare:nTF { 0 <= \l__template_tmp_int <= 9 }
192       {
193         \msg_info:nnnV { template } { declare-type }
194           {#1} \l__template_tmp_int
195         \prop_gput:NnV \g__template_type_prop {#1}
196           \l__template_tmp_int
197       }
198       {
199         \msg_error:nnnV { template } { bad-number-of-arguments }
200           {#1} \l__template_tmp_int
201       }
202   }

```

(End of definition for __template_define_type:nn and __template_declare_type:nn.)

12.5 Design part of template declaration

The “design” part of a template declaration defines the general behaviour of each key, and possibly a default value. However, it does not include the implementation. This means that what happens here is the two properties are saved to appropriate lists, which can then be used later to recover the information when implementing the keys.

`__template_declare_template_keys:nnnn` The main function for the “design” part of creating a template starts by checking that the type exists and that the number of arguments required agree. If that is all fine, then the two storage areas for defaults and keytypes are initialised. The mechanism is then set up for the `l3keys` module to actually parse the keys. Finally, the code hands off to the storage routine to save the parsed information properly.

```

203 \cs_new_protected:Npn \__template_declare_template_keys:nnnn #1#2#3#4
204   {
205     \__template_execute_if_type_exist:nT {#1}
206     {
207       \__template_execute_if_arg_agree:nnT {#1} {#3}
208       {
209         \prop_clear:N \l__template_values_prop

```



```

210         \prop_clear:N \l__template_keytypes_prop
211         \seq_clear:N \l__template_key_order_seq
212         \keyval_parse:NNn
213         \__template_parse_keys_elt:n \__template_parse_keys_elt:nn {#4}
214         \__template_store_defaults:nn {#1} {#2}
215         \__template_store_keytypes:nn {#1} {#2}
216     }
217 }
218 }

```

(End of definition for `__template_declare_template_keys:nnnn`.)

`__template_parse_keys_elt:n` Processing the key part of the key–value pair is always carried out using this function, even if a value was found. First, the key name is separated from the keytype, and if necessary the keytype is separated into two parts. This information is then used to check that the keytype is valid, before storing the keytype (plus argument if necessary) as a property of the key name. The key name is also stored (in braces) in the token list to record the order the keys are defined in.

```

219 \cs_new_protected:Npn \__template_parse_keys_elt:n #1
220 {
221     \__template_split_keytype:n {#1}
222     \bool_if:NF \l__template_error_bool
223     {
224         \__template_execute_if_keytype_exist:VT \l__template_keytype_tl
225         {
226             \seq_map_function:NN \c__template_keytypes_arg_seq
227             \__template_parse_keys_elt_aux:n
228             \bool_if:NF \l__template_error_bool
229             {
230                 \seq_if_in:NoTF \l__template_key_order_seq
231                 \l__template_key_name_tl
232                 {
233                     \msg_error:nnV { template } { duplicate-key-interface }
234                     \l__template_key_name_tl
235                 }
236                 { \__template_parse_keys_elt_aux: }
237             }
238         }
239     }
240 }
241 \cs_new_protected:Npn \__template_parse_keys_elt_aux:n #1
242 {
243     \str_if_eq:VnT \l__template_keytype_tl {#1}
244     {
245         \tl_if_empty:NT \l__template_keytype_arg_tl
246         {
247             \msg_error:nnn { template } { keytype-requires-argument } {#1}
248             \bool_set_true:N \l__template_error_bool
249             \seq_map_break:
250         }
251     }
252 }
253 \cs_new_protected:Npn \__template_parse_keys_elt_aux:
254 {

```

```

255 \tl_set:Ne \l__template_tmp_tl
256 {
257   \l__template_keytype_tl
258   \tl_if_empty:NF \l__template_keytype_arg_tl
259   { { \l__template_keytype_arg_tl } }
260 }
261 \prop_put:NVV \l__template_keytypes_prop \l__template_key_name_tl
262 \l__template_tmp_tl
263 \seq_put_right:NV \l__template_key_order_seq \l__template_key_name_tl
264 \str_if_eq:VnT \l__template_keytype_tl { choice }
265 {
266   \clist_if_in:NnT \l__template_keytype_arg_tl { unknown }
267   { \msg_error:nn { template } { choice-unknown-reserved } }
268 }
269 }

```

(End of definition for `__template_parse_keys_elt:n`, `__template_parse_keys_elt_aux:n`, and `__template_parse_keys_elt_aux:.`)

`__template_parse_keys_elt:nn` For keys which have a default, the keytype and key name are first separated out by the `__template_parse_keys_elt:n` routine, before storing the default value in the scratch property list.

```

270 \cs_new_protected:Npn \__template_parse_keys_elt:nn #1#2
271 {
272   \__template_parse_keys_elt:n {#1}
273   \use:c { __template_store_value_ \l__template_keytype_tl :n } {#2}
274 }

```

(End of definition for `__template_parse_keys_elt:nn`.)

`__template_split_keytype:n` The keytype and key name should be separated by `:`. As the definition might be given inside or outside of a code block, the category code of colons is standardised. After that, the standard delimited argument method is used to separate the two parts.

`__template_split_keytype_aux:w`

```

275 \cs_new_protected:Npe \__template_split_keytype:n #1
276 {
277   \exp_not:N \bool_set_false:N \exp_not:N \l__template_error_bool
278   \tl_set:Nn \exp_not:N \l__template_tmp_tl {#1}
279   \tl_replace_all:Nnn \exp_not:N \l__template_tmp_tl { : } { \token_to_str:N : }
280   \tl_if_in:VnTF \exp_not:N \l__template_tmp_tl { \token_to_str:N : }
281   {
282     \exp_not:n
283     {
284       \tl_clear:N \l__template_key_name_tl
285       \exp_after:wN \__template_split_keytype_aux:w
286       \l__template_tmp_tl \s__template_stop
287     }
288   }
289   {
290     \exp_not:N \bool_set_true:N \exp_not:N \l__template_error_bool
291     \msg_error:nnn { template } { missing-keytype } {#1}
292   }
293 }
294 \use:e
295 {

```

```

296 \cs_new_protected:Npn \exp_not:N \__template_split_keytype_aux:w
297 #1 \token_to_str:N : #2 \s__template_stop
298 {
299   \tl_put_right:Ne \exp_not:N \l__template_key_name_tl
300   {
301     \exp_not:N \tl_trim_spaces:e
302     { \exp_not:N \tl_to_str:n {#1} }
303   }
304   \tl_if_in:nnTF {#2} { \token_to_str:N : }
305   {
306     \tl_put_right:Nn \exp_not:N \l__template_key_name_tl
307     { \token_to_str:N : }
308     \exp_not:N \__template_split_keytype_aux:w #2 \s__template_stop
309   }
310   {
311     \exp_not:N \tl_if_empty:NTF \exp_not:N \l__template_key_name_tl
312     {
313       \msg_error:nnn { template } { empty-key-name }
314       { \token_to_str:N : #2 }
315     }
316     { \exp_not:N \__template_split_keytype_arg:n {#2} }
317   }
318 }
319 }

```

(End of definition for __template_split_keytype:n and __template_split_keytype_aux:w.)

__template_split_keytype_arg:n
 __template_split_keytype_arg:V
 __template_split_keytype_arg_aux:n
 __template_split_keytype_arg_aux:w

The second stage of sorting out the keytype is to check for an argument. As there is no convenient delimiting token to look for, a check is made instead for each possible text value for the keytype. To keep things faster, this only involves the keytypes that need an argument. If a match is made, then a check is also needed to see that it is at the start of the keytype information. All being well, the split can then be applied. Any non-matching keytypes are assumed to be “correct” as given, and are left alone (this is checked by other code).

```

320 \cs_new_protected:Npn \__template_split_keytype_arg:n #1
321 {
322   \tl_set:Ne \l__template_keytype_tl { \tl_trim_spaces:n {#1} }
323   \tl_clear:N \l__template_keytype_arg_tl
324   \cs_set_protected:Npn \__template_split_keytype_arg_aux:n ##1
325   {
326     \tl_if_in:nnT {#1} {##1}
327     {
328       \cs_set:Npn \__template_split_keytype_arg_aux:w
329       ####1 ##1 ####2 \s__template_stop
330       {
331         \tl_if_blank:nT {####1}
332         {
333           \tl_set:Ne \l__template_keytype_tl
334           { \tl_trim_spaces:n {##1} }
335           \tl_if_blank:nF {####2}
336           {
337             \tl_set:Ne \l__template_keytype_arg_tl
338             { \use:n {####2} }
339           }

```

```

340         \seq_map_break:
341     }
342 }
343 \__template_split_keytype_arg_aux:w #1 \s__template_stop
344 }
345 }
346 \seq_map_function:NN \c__template_keytypes_arg_seq
347 \__template_split_keytype_arg_aux:n
348 }
349 \cs_generate_variant:Nn \__template_split_keytype_arg:n { V }
350 \cs_new:Npn \__template_split_keytype_arg_aux:n #1 { }
351 \cs_new:Npn \__template_split_keytype_arg_aux:w #1 \s__template_stop { }

```

(End of definition for `__template_split_keytype_arg:n`, `__template_split_keytype_arg_aux:n`, and `__template_split_keytype_arg_aux:w`.)

12.5.1 Storing values

As `ltemplates` pre-processes key values for efficiency reasons, there is a need to convert the values given as defaults into “ready to use” data. The same general idea is true when an instance is declared. However, assignments are not made until an instance is used, and so there has to be some intermediate storage. Furthermore, the ability to delay evaluation of results is needed. To achieve these aims, a series of “process and store” functions are defined here.

All of the information about the key (the key name and the keytype) is already stored as variables. The same property list is always used to store the data, meaning that the only argument required is the value to be processed and potentially stored.

`__template_store_value_boolean:n`

```

352 \cs_new_protected:Npn \__template_store_value_boolean:n #1
353 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#1} }

```

(End of definition for `__template_store_value_boolean:n`.)

`__template_store_value:n`

With no need to worry about delayed evaluation, these keytypes all just store the input directly.

`__template_store_value_choice:n`

`__template_store_value_function:n`

`__template_store_value_instance:n`

```

354 \cs_new_protected:Npn \__template_store_value:n #1
355 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#1} }
356 \cs_new_eq:NN \__template_store_value_choice:n \__template_store_value:n
357 \cs_new_eq:NN \__template_store_value_function:n \__template_store_value:n
358 \cs_new_eq:NN \__template_store_value_instance:n \__template_store_value:n

```

(End of definition for `__template_store_value:n` and others.)

`__template_store_value_aux:Nn`

Storing values in `\l__template_values_prop` is in most cases the same.

`__template_store_value_integer:n`

`__template_store_value_length:n`

`__template_store_value_muskip:n`

`__template_store_value_real:n`

`__template_store_value_skip:n`

`__template_store_value_tokenlist:n`

`__template_store_value_commalist:n`

```

359 \cs_new_protected:Npn \__template_store_value_aux:Nn #1#2
360 { \prop_put:Non \l__template_values_prop \l__template_key_name_tl {#2} }
361 \cs_new_protected:Npn \__template_store_value_integer:n
362 { \__template_store_value_aux:Nn \int_eval:n }
363 \cs_new_protected:Npn \__template_store_value_length:n
364 { \__template_store_value_aux:Nn \dim_eval:n }
365 \cs_new_protected:Npn \__template_store_value_muskip:n
366 { \__template_store_value_aux:Nn \muskip_eval:n }
367 \cs_new_protected:Npn \__template_store_value_real:n

```

```

368 { \_template_store_value_aux:Nn \fp_eval:n }
369 \cs_new_protected:Npn \_template_store_value_skip:n
370 { \_template_store_value_aux:Nn \skip_eval:n }
371 \cs_new_protected:Npn \_template_store_value_tokenlist:n
372 { \_template_store_value_aux:Nn \use:n }
373 \cs_new_eq:NN \_template_store_value_commalist:n \_template_store_value_tokenlist:n

```

(End of definition for _template_store_value_aux:Nn and others.)

12.6 Implementation part of template declaration

_template_declare_template_code:nnnn
_template_declare_template_code:nnnn

The main function for implementing a template starts with a couple of simple checks to make sure that there are no obvious mistakes: the number of arguments must agree and the template keys must have been declared.

```

374 \cs_new_protected:Npn \_template_declare_template_code:nnnn #1#2#3#4#5
375 {
376   \_template_execute_if_type_exist:nT {#1}
377   {
378     \_template_execute_if_arg_agree:nnT {#1} {#3}
379     {
380       \_template_if_keys_exist:nnT {#1} {#2}
381       {
382         \_template_store_key_implementation:nnn {#1} {#2} {#4}
383         \regex_match:nnTF { \c { AssignTemplateKeys } } {#5}
384         { \_template_declare_template_code:nnnn {#1} {#2} {#3} {#5} }
385         {
386           \_template_declare_template_code:nnnn
387             {#1} {#2} {#3} { \AssignTemplateKeys #5 }
388         }
389       }
390     }
391   }
392 }
393 \cs_new_protected:Npn \_template_declare_template_code:nnnn #1#2#3#4
394 {
395   \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
396   { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
397   \cs_generate_from_arg_count:cNnn
398   { \c__template_code_root_tl #1 / #2 }
399   \cs_gset_protected:Npn {#3} {#4}
400 }

```

(End of definition for _template_declare_template_code:nnnn and _template_declare_template_code:nnnn.)

_template_store_key_implementation:nnn

Actually storing the implementation part of a template is quite easy as it only requires the list of keys given to be turned into a property list. There is also some error-checking to do, hence the need to have the list of defined keytypes available. In certain cases (when choices are involved) parsing the key results in changes to the default values. That is why they are loaded and then saved again.

```

401 \cs_new_protected:Npn \_template_store_key_implementation:nnn #1#2#3
402 {
403   \_template_recover_defaults:nn {#1} {#2}
404   \_template_recover_keytypes:nn {#1} {#2}

```

```

405 \prop_clear:N \l__template_vars_prop
406 \keyval_parse:nnn
407   { \__template_parse_vars_elt:n } { \__template_parse_vars_elt:nnn { #1 / #2 } } {#3}
408 \__template_store_vars:nn {#1} {#2}
409 \prop_map_inline:Nn \l__template_keytypes_prop
410   { \msg_error:nnnnn { template } { key-not-implemented } {##1} {#2} {#1} }
411 }

```

(End of definition for `__template_store_key_implementation:nnn`.)

`__template_parse_vars_elt:n` At the implementation stage, every key must have a value given. So this is an error function.

```

412 \cs_new_protected:Npn \__template_parse_vars_elt:n #1
413   { \msg_error:nnn { template } { key-no-variable } {#1} }

```

(End of definition for `__template_parse_vars_elt:n`.)

`__template_parse_vars_elt:nnn` The actual storage part here is very simple: the storage bin name is placed into the property list. At the same time, a comparison is made with the keytypes defined earlier: if there is a mismatch then an error is raised.

```

414 \cs_new_protected:Npn \__template_parse_vars_elt:nnn #1#2#3
415   {
416     \tl_set:Ne \l__template_key_name_tl
417       { \tl_trim_spaces:e { \tl_to_str:n {#2} } }
418     \prop_get:NVNTF \l__template_keytypes_prop
419       \l__template_key_name_tl
420       \l__template_keytype_tl
421       {
422         \__template_split_keytype_arg:V \l__template_keytype_tl
423         \__template_parse_vars_elt_aux:nn {#1} {#3}
424         \prop_remove:NV \l__template_keytypes_prop \l__template_key_name_tl
425       }
426     { \msg_error:nnn { template } { unknown-key } {#2} }
427   }

```

Split off any leading global and they look for the way to implement.

```

428 \cs_new_protected:Npn \__template_parse_vars_elt_aux:nn #1#2
429   {
430     \__template_parse_vars_elt_aux:nw {#1} #2 global global \s__template_stop
431   }
432 \cs_new_protected:Npn \__template_parse_vars_elt_aux:nw
433   #1#2 global #3 global #4 \s__template_stop
434   {
435     \tl_if_blank:nTF {#4}
436     { \__template_parse_vars_elt_aux:nnn {#1} { } {#2} }
437     {
438       \tl_if_blank:nTF {#2}
439       {
440         \__template_parse_vars_elt_aux:nne
441           {#1} { global } { \tl_trim_spaces:n {#3} }
442       }
443       { \msg_error:nnn { template } { bad-variable } { #2 global #3 } }
444     }
445   }
446 \cs_new_protected:Npn \__template_parse_vars_elt_aux:nnn #1#2#3

```

```

447 {
448   \str_case:VnF \l__template_keytype_tl
449   {
450     { choice } { \__template_implement_choices:nn {#1} {#3} }
451     { function }
452     {
453       \cs_if_exist:NF #3
454       { \cs_new:Npn #3 { } }
455       \__template_parse_vars_elt_key:nn {#1}
456       {
457         .code:n =
458         {
459           \cs_generate_from_arg_count:NNnn
460           \exp_not:N #3
461           \exp_not:c
462           { cs_ \str_if_eq:nnT {#1} { global } { g } set:Npn }
463           { \exp_not:V \l__template_keytype_arg_tl }
464           {##1}
465         }
466       }
467       \prop_put:NVn \l__template_vars_prop
468       \l__template_key_name_tl {#2#3}
469     }
470     { instance }
471     {
472       \__template_parse_vars_elt_key:nn {#1}
473       {
474         .code:n =
475         {
476           \exp_not:c
477           { cs_ \str_if_eq:nnT {#1} { global } { g } set:Npn }
478           \exp_not:N #3 { \UseInstance {##1} }
479         }
480       }
481       \prop_put:NVn \l__template_vars_prop
482       \l__template_key_name_tl {#2#3}
483     }
484   }
485   {
486     \tl_if_single:nTF {#3}
487     {
488       \cs_if_exist:NF #3
489       { \use:c { \__template_map_var_type: _new:N } #3 }
490       \__template_parse_vars_elt_key:nn {#1}
491       {
492         . \__template_map_var_type:
493         _ \str_if_eq:nnT {#1} { global } { g } set:N
494         = \exp_not:N #3
495       }
496       \prop_put:NVn \l__template_vars_prop
497       \l__template_key_name_tl {#2#3}
498     }
499     { \msg_error:nnn { template } { bad-variable } {#2#3} }
500   }

```

```

501 }
502 \cs_generate_variant:Nn \__template_parse_vars_elt_aux:nnn { nne }
503 \cs_new_protected:Npn \__template_parse_vars_elt_key:nn #1#2
504 {
505   \keys_define:ne { template / #1 }
506     { \l__template_key_name_tl #2 }
507 }

```

(End of definition for __template_parse_vars_elt:nnn and others.)

__template_map_var_type: Turn a “friendly” variable type into an expl3 one.

```

508 \cs_new:Npn \__template_map_var_type:
509 {
510   \str_case:Vn \l__template_keytype_tl
511   {
512     { boolean } { bool }
513     { commalist } { clist }
514     { integer } { int }
515     { length } { dim }
516     { muskip } { muskip }
517     { real } { fp }
518     { skip } { skip }
519     { tokenlist } { tl }
520   }
521 }

```

(End of definition for __template_map_var_type:.)

__template_implement_choices:nn Implementing choices requires a second key–value loop. So after a little set-up, the standard parser is called.

__template_implement_choices_default:

```

522 \cs_new_protected:Npn \__template_implement_choices:nn #1#2
523 {
524   \clist_set:NV \l__template_tmp_clist \l__template_keytype_arg_tl
525   \prop_put:NVn \l__template_vars_prop \l__template_key_name_tl { }
526   \keys_define:ne { template / #1 } { \l__template_key_name_tl .choice: }
527   \keyval_parse:nnn
528     { \__template_implement_choice_elt:n }
529     { \__template_implement_choice_elt:nnn {#1} }
530     {#2}
531   \prop_get:NVNT \l__template_values_prop \l__template_key_name_tl
532     \l__template_tmp_tl
533   { \__template_implement_choices_default: }
534   \clist_if_empty:NF \l__template_tmp_clist
535   {
536     \clist_map_inline:Nn \l__template_tmp_clist
537       { \msg_error:nnn { template } { choice-not-implemented } {##1} }
538   }
539 }

```

A sanity check for the default value, so that an error is raised now and not when converting to assignments.

```

540 \cs_new_protected:Npn \__template_implement_choices_default:
541 {
542   \tl_set:Ne \l__template_tmp_tl
543     { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }

```



```

544 \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
545 {
546   \tl_set:Ne \l__template_tmp_tl
547   { \l__template_key_name_tl \c_space_tl \l__template_tmp_tl }
548   \prop_if_in:NVF \l__template_vars_prop \l__template_tmp_tl
549   {
550     \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
551     \l__template_tmp_tl
552     \__template_split_keytype_arg:V \l__template_tmp_tl
553     \prop_get:NVN \l__template_values_prop \l__template_key_name_tl
554     \l__template_tmp_tl
555     \msg_error:nnVV { template } { unknown-default-choice }
556     \l__template_key_name_tl
557     \l__template_key_name_tl
558   }
559 }
560 }

```

(End of definition for `__template_implement_choices:nn` and `__template_implement_choices_default:.`)

```

\__template_implement_choice_elt:nnn
\__template_implement_choice_elt_aux:nnn
\__template_implement_choice_elt_aux:n
\__template_implement_choice_elt:n

```

The actual storage of the implementation of a choice is mainly about error checking. The code here ensures that all choices have to have been declared, apart from the special **unknown** choice, which must come last. The code for each choice is stored along with the key name in the variables property list.

```

561 \cs_new_protected:Npn \__template_implement_choice_elt:nnn #1#2#3
562 {
563   \clist_if_empty:NTF \l__template_tmp_clist
564   {
565     \str_if_eq:nnTF {#2} { unknown }
566     { \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3} }
567     { \__template_implement_choice_elt_aux:n {#2} }
568   }
569   {
570     \clist_if_in:NnTF \l__template_tmp_clist {#2}
571     {
572       \clist_remove_all:Nn \l__template_tmp_clist {#2}
573       \__template_implement_choice_elt_aux:nnn {#1} {#2} {#3}
574     }
575     { \__template_implement_choice_elt_aux:n {#2} }
576   }
577 }
578 \cs_new_protected:Npn \__template_implement_choice_elt_aux:n #1
579 {
580   \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
581   \l__template_tmp_tl
582   \__template_split_keytype_arg:V \l__template_tmp_tl
583   \msg_error:nnVn { template } { unknown-choice } \l__template_key_name_tl {#1}
584 }
585 \cs_new_protected:Npn \__template_implement_choice_elt_aux:nnn #1#2#3
586 {
587   \keys_define:ne { template / #1 }
588   { \l__template_key_name_tl / #2 .code:n = { \exp_not:n {#3} } }
589   \tl_set:Ne \l__template_tmp_tl

```

```

590     { \l__template_key_name_tl \c_space_tl #2 }
591     \prop_put:NVn \l__template_vars_prop \l__template_tmp_tl {#3}
592   }
593 \cs_new_protected:Npn \__template_implement_choice_elt:n #1
594   {
595     \msg_error:nnVn { template } { choice-requires-code }
596     \l__template_key_name_tl {#1}
597   }

```

(End of definition for __template_implement_choice_elt:nnn and others.)

12.7 Editing template defaults

`__template_edit_defaults:nnn` Editing the template defaults means getting the values back out of the store, then parsing the list of new values before putting the updated list back into storage.

```

598 \cs_new_protected:Npn \__template_edit_defaults:nnn #1#2#3
599   {
600     \__template_if_keys_exist:nnT {#1} {#2}
601     {
602       \__template_recover_defaults:nn {#1} {#2}
603       \__template_parse_values:nnn {#1} {#2} {#3}
604       \__template_store_defaults:nn {#1} {#2}
605     }
606   }

```

(End of definition for __template_edit_defaults:nnn.)

`__template_parse_values:nnn` The routine to parse values is the same for both editing a template and setting up an instance. So the code here does only the minimum necessary for reading the values.

```

607 \cs_new_protected:Npn \__template_parse_values:nnn #1#2#3
608   {
609     \__template_recover_keytypes:nn {#1} {#2}
610     \keyval_parse:NNn
611     \__template_parse_values_elt:n \__template_parse_values_elt:nn {#3}
612   }

```

(End of definition for __template_parse_values:nnn.)

`__template_parse_values_elt:n` Every key needs a value, so this is just an error routine.

```

613 \cs_new_protected:Npn \__template_parse_values_elt:n #1
614   {
615     \bool_set_true:N \l__template_error_bool
616     \msg_error:nnn { template } { key-no-value } {#1}
617   }

```

(End of definition for __template_parse_values_elt:n.)

`__template_parse_values_elt:nn`
`__template_parse_values_elt_aux:n` To store the value, find the keytype then call the saving function. These need the current key name saved as `\l__template_key_name_tl`.

```

618 \cs_new_protected:Npn \__template_parse_values_elt:nn #1#2
619   {
620     \tl_set:Ne \l__template_key_name_tl
621     { \tl_trim_spaces:e { \tl_to_str:n {#1} } }
622     \prop_get:NVNTF \l__template_keytypes_prop \l__template_key_name_tl

```

```

623     \l__template_tmp_tl
624     { \__template_parse_values_elt_aux:n {#2} }
625     { \msg_error:nnV { template } { unknown-key } \l__template_key_name_tl }
626   }
627 \cs_new_protected:Npn \__template_parse_values_elt_aux:n #1
628 {
629   \__template_split_keytype_arg:V \l__template_tmp_tl
630   \use:c { __template_store_value_ \l__template_keytype_tl :n } {#1}
631 }

```

(End of definition for `__template_parse_values_elt:nn` and `__template_parse_values_elt_aux:n`.)

`__template_template_set_eq:nnn`

To copy a template, each of the lists plus the code has to be copied across. To keep this independent of the list storage system, it is all done with two-part shuffles.

```

632 \cs_new_protected:Npn \__template_template_set_eq:nnn #1#2#3
633 {
634   \__template_recover_defaults:nn {#1} {#3}
635   \__template_store_defaults:nn {#1} {#2}
636   \__template_recover_keytypes:nn {#1} {#3}
637   \__template_store_keytypes:nn {#1} {#2}
638   \__template_recover_vars:nn {#1} {#3}
639   \__template_store_vars:nn {#1} {#2}
640   \cs_if_exist:cT { \c__template_code_root_tl #1 / #2 }
641   { \msg_info:nnnn { template } { declare-template-code } {#1} {#2} }
642   \cs_gset_eq:cc { \c__template_code_root_tl #1 / #2 }
643   { \c__template_code_root_tl #1 / #3 }
644 }

```

(End of definition for `__template_template_set_eq:nnn`.)

12.8 Creating instances of templates

`__template_declare_instance:nnnn`
`__template_declare_instance_aux:nnnn`

Making an instance has two distinct parts. First, the keys given are parsed to transfer the values into the structured data format used internally. This allows the default and given values to be combined with no repetition. In the second step, the structured data is converted to pre-defined variable assignments, and these are stored in the function for the instance.

```

645 \cs_new_protected:Npn \__template_declare_instance:nnnn #1#2#3#4
646 {
647   \__template_execute_if_code_exist:nnT {#1} {#2}
648   {
649     \__template_recover_defaults:nn {#1} {#2}
650     \__template_recover_vars:nn {#1} {#2}
651     \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
652   }
653 }
654 \cs_new_protected:Npn \__template_declare_instance_aux:nnnn #1#2#3#4
655 {
656   \bool_set_false:N \l__template_error_bool
657   \__template_parse_values:nnn {#1} {#2} {#4}
658   \bool_if:NF \l__template_error_bool
659   {
660     \prop_put:Nnn \l__template_values_prop { from-template } {#2}
661     \__template_store_values:nn {#1} {#3}

```

```

662     \__template_convert_to_assignments:
663     \cs_if_exist:cT { \c__template_instances_root_tl #1 / #3 }
664       { \msg_info:nnnn { template } { declare-instance } {#3} {#1} }
665     \cs_set_protected:cpe { \c__template_instances_root_tl #1 / #3 }
666       {
667         \exp_not:N \__template_assignments_push:n
668           { \exp_not:V \l__template_assignments_tl }
669         \exp_not:c { \c__template_code_root_tl #1 / #2 }
670       }
671   }
672 }

```

(End of definition for __template_declare_instance:nnnn and __template_declare_instance_aux:nnnn.)

__template_instance_set_eq:nnn Copy-paste an instance.

```

673 \cs_new_protected:Npn \__template_instance_set_eq:nnn #1#2#3
674 {
675   \__template_if_instance_exist:nnTF {#1} {#3}
676   {
677     \__template_recover_values:nn {#1} {#3}
678     \__template_store_values:nn {#1} {#2}
679     \cs_if_exist:cT { \c__template_instances_root_tl #1 / #2 }
680       { \msg_info:nnnn { template } { declare-instance } {#2} {#1} }
681     \cs_set_eq:cc { \c__template_instances_root_tl #1 / #2 }
682       { \c__template_instances_root_tl #1 / #3 }
683   }
684   { \msg_error:nnnn { template } { unknown-instance } {#1} {#3} }
685 }

```

(End of definition for __template_instance_set_eq:nnn.)

__template_edit_instance:nnn Editing an instance is almost identical to declaring one. The only variation is the source of the values to use. When editing, they are recovered from the previous instance run.

__template_edit_instance_aux:nnnn

__template_edit_instance_aux:nVnn

```

686 \cs_new_protected:Npn \__template_edit_instance:nnn #1#2#3
687 {
688   \__template_if_instance_exist:nnTF {#1} {#2}
689   {
690     \__template_recover_values:nn {#1} {#2}
691     \prop_get:NnN \l__template_values_prop { from-template }
692       \l__template_tmp_tl
693     \__template_edit_instance_aux:nVnn
694       {#1} \l__template_tmp_tl {#2} {#3}
695   }
696   { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
697 }
698 \cs_new_protected:Npn \__template_edit_instance_aux:nnnn #1#2#3#4
699 {
700   \__template_recover_vars:nn {#1} {#2}
701   \__template_declare_instance_aux:nnnn {#1} {#2} {#3} {#4}
702 }
703 \cs_generate_variant:Nn \__template_edit_instance_aux:nnnn { nV }

```

(End of definition for __template_edit_instance:nnn and __template_edit_instance_aux:nnnn.)

```

  \_template_convert_to_assignments:
  \_template_convert_to_assignments_aux:n
  \_template_convert_to_assignments_aux:nn
  \_template_convert_to_assignments_aux:nV

```

The idea on converting to a set of assignments is to loop over each key, so that the loop order follows the declaration order of the keys. This is done using a sequence as property lists are not “ordered”.

```

704 \cs_new_protected:Npn \_template_convert_to_assignments:
705 {
706   \tl_clear:N \l__template_assignments_tl
707   \seq_map_function:NN \l__template_key_order_seq
708     \_template_convert_to_assignments_aux:n
709 }
710 \cs_new_protected:Npn \_template_convert_to_assignments_aux:n #1
711 {
712   \prop_get:NnN \l__template_keytypes_prop {#1} \l__template_tmp_tl
713   \_template_convert_to_assignments_aux:nV {#1} \l__template_tmp_tl
714 }

```

The second auxiliary function actually does the work. The arguments here are the key name (#1) and the keytype (#2). From those, the value to assign and the name of the appropriate variable are recovered. A bit of work is then needed to sort out keytypes with arguments (for example instances), and to look for global assignments. Once that is done, a hand-off can be made to the handler for the relevant keytype.

```

715 \cs_new_protected:Npn \_template_convert_to_assignments_aux:nn #1#2
716 {
717   \prop_get:NnNT \l__template_values_prop {#1} \l__template_value_tl
718   {
719     \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_var_tl
720     {
721       \_template_split_keytype_arg:n {#2}
722       \str_if_eq:VnF \l__template_keytype_tl { choice }
723       {
724         \str_if_eq:VnF \l__template_keytype_tl { code }
725         { \_template_find_global: }
726       }
727       \tl_set:Nn \l__template_key_name_tl {#1}
728       \cs_if_exist_use:cF { __template_assign_ \l__template_keytype_tl : }
729       { \_template_assign_variable: }
730     }
731     { \msg_error:nnn { template } { unknown-attribute } {#1} }
732   }
733 }
734 \cs_generate_variant:Nn \_template_convert_to_assignments_aux:nn { nV }

```

(End of definition for _template_convert_to_assignments:, _template_convert_to_assignments_aux:n, and _template_convert_to_assignments_aux:nn.)

```

\_template_find_global:
  \_template_find_global_aux:w

```

Global assignments should have the phrase global at the front. This is pretty easy to find: no other error checking, though.

```

735 \cs_new_protected:Npn \_template_find_global:
736 {
737   \bool_set_false:N \l__template_global_bool
738   \tl_if_in:onT \l__template_var_tl { global }
739   {
740     \exp_after:wN \_template_find_global_aux:w \l__template_var_tl \s__template_stop
741   }
742 }

```

```

743 \cs_new_protected:Npn \__template_find_global_aux:w #1 global #2 \s__template_stop
744 {
745   \tl_set:Nn \l__template_var_tl {#2}
746   \bool_set_true:N \l__template_global_bool
747 }

```

(End of definition for `__template_find_global:` and `__template_find_global_aux:w`.)

12.9 Using templates directly

`__template_use_template:nnn` Directly use a template with a particular parameter setting. This is also picked up if used in a nested fashion inside a parameter list. The idea is essentially the same as creating an instance, just with no saving of the result.

```

748 \cs_new_protected:Npn \__template_use_template:nnn #1#2#3
749 {
750   \__template_execute_if_code_exist:nnT {#1} {#2}
751   {
752     \__template_recover_defaults:nn {#1} {#2}
753     \__template_recover_vars:nn {#1} {#2}
754     \__template_parse_values:nnn {#1} {#2} {#3}
755     \__template_convert_to_assignments:
756     \use:c { \c__template_code_root_tl #1 / #2 }
757   }
758 }

```

(End of definition for `__template_use_template:nnn`.)

12.10 Assigning values to variables

`__template_assign_boolean:` Setting a Boolean value is slightly different to everything else as the value can be used to work out which `set` function to call. As long as there is no need to recover things from another variable, everything is pretty easy. If there is, then we need to allow for the fact that the recovered value here will *not* be expandable, so needs to be converted to something that is.

```

759 \cs_new_protected:Npn \__template_assign_boolean:
760 {
761   \bool_if:NTF \l__template_global_bool
762   { \__template_assign_boolean_aux:n { bool_gset } }
763   { \__template_assign_boolean_aux:n { bool_set } }
764 }
765 \cs_new_protected:Npn \__template_assign_boolean_aux:n #1
766 {
767   \__template_if_key_value:VTF \l__template_value_tl
768   {
769     \__template_key_to_value:
770     \tl_put_right:Ne \l__template_assignments_tl
771     {
772       \exp_not:c { #1 _eq:NN }
773       \exp_not:V \l__template_var_tl
774       \exp_not:V \l__template_value_tl
775     }
776   }
777 }

```

```

778     \tl_put_right:Ne \l__template_assignments_tl
779     {
780         \exp_not:c { #1 _ \l__template_value_tl :N }
781         \exp_not:V \l__template_var_tl
782     }
783 }
784 }

```

(End of definition for __template_assign_boolean: and __template_assign_boolean_aux:n.)

__template_assign_choice: The idea here is to find either the choice as-given or else the special unknown choice, and to copy the appropriate code across.

```

\__template_assign_choice_aux:nF
\__template_assign_choice_aux:eF
785 \cs_new_protected:Npn \__template_assign_choice:
786 {
787     \__template_assign_choice_aux:eF
788     { \l__template_key_name_tl \c_space_tl \l__template_value_tl }
789     {
790         \__template_assign_choice_aux:eF
791         { \l__template_key_name_tl \c_space_tl unknown }
792         {
793             \prop_get:NVN \l__template_keytypes_prop \l__template_key_name_tl
794             \l__template_tmp_tl
795             \__template_split_keytype_arg:V \l__template_tmp_tl
796             \msg_error:nnVV { template } { unknown-choice }
797             \l__template_key_name_tl
798             \l__template_value_tl
799         }
800     }
801 }
802 \cs_new_protected:Npn \__template_assign_choice_aux:nF #1
803 {
804     \prop_get:NnNTF \l__template_vars_prop {#1} \l__template_tmp_tl
805     { \tl_put_right:NV \l__template_assignments_tl \l__template_tmp_tl }
806 }
807 \cs_generate_variant:Nn \__template_assign_choice_aux:nF { e }

```

(End of definition for __template_assign_choice: and __template_assign_choice_aux:nF.)

__template_assign_function: This looks a bit messy but is only actually one function.

```

\__template_assign_function_aux:N
808 \cs_new_protected:Npn \__template_assign_function:
809 {
810     \bool_if:NTF \l__template_global_bool
811     { \__template_assign_function_aux:N \cs_gset:Npn }
812     { \__template_assign_function_aux:N \cs_set:Npn }
813 }
814 \cs_new_protected:Npn \__template_assign_function_aux:N #1
815 {
816     \tl_put_right:Ne \l__template_assignments_tl
817     {
818         \cs_generate_from_arg_count:NNnn
819         \exp_not:V \l__template_var_tl
820         \exp_not:N #1
821         { \exp_not:V \l__template_keytype_arg_tl }
822         { \exp_not:V \l__template_value_tl }

```

```

823     }
824 }

```

(End of definition for `__template_assign_function:` and `__template_assign_function_aux:N.`)

`__template_assign_instance:` Using an instance means adding the appropriate function creation to the tl. No checks
`__template_assign_instance_aux:N` are made at this stage, so if the instance is not valid then errors will arise later.

```

825 \cs_new_protected:Npn \__template_assign_instance:
826 {
827   \bool_if:NTF \l__template_global_bool
828     { \__template_assign_instance_aux:N \cs_gset_protected:Npn }
829     { \__template_assign_instance_aux:N \cs_set_protected:Npn }
830 }
831 \cs_new_protected:Npn \__template_assign_instance_aux:N #1
832 {
833   \tl_put_right:Ne \l__template_assignments_tl
834   {
835     \exp_not:N #1 \exp_not:V \l__template_var_tl
836     {
837       \__template_use_instance:nn
838       { \exp_not:V \l__template_keytype_arg_tl }
839       { \exp_not:V \l__template_value_tl }
840     }
841   }
842 }

```

(End of definition for `__template_assign_instance:` and `__template_assign_instance_aux:N.`)

`__template_assign_variable:` A general-purpose function for all of the other assignments. As long as the value is not
`__template_assign_variable:n` coming from another variable, the stored value is simply transferred for output. We use V-type expansion for the `\KeyValue` case: for token lists this is essential, whilst for register-based variables, it does no harm and avoids needing a low-level test.

```

843 \cs_new_protected:Npn \__template_assign_variable:
844 {
845   \exp_args:Ne \__template_assign_variable:n
846   {
847     \__template_map_var_type:
848     -
849     \bool_if:NT \l__template_global_bool { g }
850     set:N
851   }
852 }
853 \cs_new_protected:Npn \__template_assign_variable:n #1
854 {
855   \__template_if_key_value:VTF \l__template_value_tl
856   {
857     \__template_key_to_value:
858     \tl_put_right:Ne \l__template_assignments_tl
859     {
860       \exp_not:c { #1 V } \exp_not:V \l__template_var_tl
861       \exp_not:V \l__template_value_tl
862     }
863   }
864 }

```



```

865     \tl_put_right:Ne \l__template_assignments_tl
866     {
867         \exp_not:c { #1 n } \exp_not:V \l__template_var_tl
868         { \exp_not:V \l__template_value_tl }
869     }
870 }
871 }

```

(End of definition for `__template_assign_variable:` and `__template_assign_variable:n`.)

`__template_key_to_value:` The idea here is to recover the attribute value of another key. To do that, the marker is removed and a look up takes place. If this is successful, then the name of the variable of the attribute is returned. This assumes that the value will be used in context where it will be converted to a value, for example when setting a number. There is also a need to check in case the copied value happens to be `global`.

```

872 \cs_new_protected:Npn \__template_key_to_value:
873 { \exp_after:wN \__template_key_to_value_auxi:w \l__template_value_tl }
874 \cs_new_protected:Npn \__template_key_to_value_auxi:w \KeyValue #1
875 {
876     \tl_set:Ne \l__template_tmp_tl { \tl_trim_spaces:e { \tl_to_str:n {#1} } }
877     \prop_get:NVNTF \l__template_vars_prop \l__template_tmp_tl
878     \l__template_value_tl
879     {
880         \exp_after:wN \__template_key_to_value_auxii:w \l__template_value_tl
881         \s__template_mark global \q__template_nil \s__template_stop
882     }
883     { \msg_error:nnV { template } { unknown-attribute } \l__template_tmp_tl }
884 }
885 \cs_new_protected:Npn \__template_key_to_value_auxii:w #1 global #2#3 \s__template_stop
886 {
887     \__template_quark_if_nil:NF #2
888     { \tl_set:Nn \l__template_value_tl {#2} }
889 }

```

(End of definition for `__template_key_to_value:`, `__template_key_to_value_auxi:w`, and `__template_key_to_value_auxii:w`.)

12.11 Using instances

`__template_use_instance:nn` Using an instance is just a question of finding the appropriate function. If nothing is found, an error is raised. One complication is that if the first token of argument #2 is `\UseTemplate` then that is also valid. There is an error-test to make sure that the types agree, and if so the template is used directly.

```

890 \cs_new_protected:Npn \__template_use_instance:nn #1#2
891 {
892     \__template_if_use_template:nTF {#2}
893     { \__template_use_instance_aux:nNnnn {#1} #2 }
894     { \__template_use_instance_aux:nn {#1} {#2} }
895 }
896 \cs_new_protected:Npn \__template_use_instance_aux:nNnnn #1#2#3#4#5
897 {
898     \str_if_eq:nnTF {#1} {#3}
899     { \__template_use_template:nnn {#3} {#4} {#5} }
900     { \msg_error:nnnn { template } { type-mismatch } {#1} {#3} }

```

```

901 }
902 \cs_new_protected:Npn \__template_use_instance_aux:nn #1#2
903 {
904   \__template_if_instance_exist:nnTF {#1} {#2}
905   { \use:c { \c__template_instances_root_tl #1 / #2 } }
906   { \msg_error:nnnn { template } { unknown-instance } {#1} {#2} }
907 }

```

(End of definition for `__template_use_instance:nn`, `__template_use_instance_aux:nNnnn`, and `__template_use_instance_aux:nn`.)

12.12 Assignment manipulation

A few functions to transfer assignments about, as this is needed by `\AssignTemplateKeys`.

```

\__template_assignments_pop: To actually use the assignments.
908 \cs_new:Npn \__template_assignments_pop: { \l__template_assignments_tl }

(End of definition for \__template_assignments_pop:.)

```

```

\__template_assignments_push:n Here, the assignments are stored for later use.
909 \cs_new_protected:Npn \__template_assignments_push:n #1
910 { \tl_set:Nn \l__template_assignments_tl {#1} }

(End of definition for \__template_assignments_push:n.)

```

12.13 Showing templates and instances

```

\__template_show_code:nn Showing the code for a template is just a translation of \cs_show:c.
911 \cs_new_protected:Npn \__template_show_code:nn #1#2
912 { \cs_show:c { \c__template_code_root_tl #1 / #2 } }

(End of definition for \__template_show_code:nn.)

```

```

\__template_show_defaults:nn A modified version of the property-list printing code, such that the output refers to
\__template_show_keytypes:nn templates and instances rather than to the underlying structures.
\__template_show_vars:nn
\__template_show:Nnnn
913 \cs_new_protected:Npn \__template_show_defaults:nn #1#2
914 {
915   \__template_if_keys_exist:nnT {#1} {#2}
916   {
917     \__template_recover_defaults:nn {#1} {#2}
918     \__template_show:Nnnn \l__template_values_prop
919     {#1} {#2} { default~values }
920   }
921 }
922 \cs_new_protected:Npn \__template_show_keytypes:nn #1#2
923 {
924   \__template_if_keys_exist:nnT {#1} {#2}
925   {
926     \__template_recover_keytypes:nn {#1} {#2}
927     \__template_show:Nnnn \l__template_keytypes_prop
928     {#1} {#2} { interface }
929   }
930 }

```

```

931 \cs_new_protected:Npn \__template_show_vars:nn #1#2
932 {
933     \__template_execute_if_code_exist:nnT {#1} {#2}
934     {
935         \__template_recover_vars:nn {#1} {#2}
936         \__template_show:Nnnn \l__template_vars_prop
937         {#1} {#2} { variable-mapping }
938     }
939 }
940 \cs_new_protected:Npn \__template_show:Nnnn #1#2#3#4
941 {
942     \msg_show:nneeee { template } { show-attribute }
943     { \tl_to_str:n {#2} }
944     { \tl_to_str:n {#3} }
945     { \tl_to_str:n {#4} }
946     { \prop_map_function:NN #1 \msg_show_item_unbraced:nn }
947 }

```

(End of definition for __template_show_defaults:nn and others.)

__template_show_values:nn Instance values are a little more complex, as is the template to consider.

```

948 \cs_new_protected:Npn \__template_show_values:nn #1#2
949 {
950     \__template_if_instance_exist:nnT {#1} {#2}
951     {
952         \__template_recover_values:nn {#1} {#2}
953         \msg_show:nneee { template } { show-values }
954         { \tl_to_str:n {#1} }
955         { \tl_to_str:n {#2} }
956         {
957             \prop_map_function:NN \l__template_values_prop
958             \msg_show_item_unbraced:nn
959         }
960     }
961 }

```

(End of definition for __template_show_values:nn.)

12.14 Messages

The text for error messages: short and long text for all of them.

```

962 \msg_new:nnnn { template } { argument-number-mismatch }
963 { Template-type~'#1'~takes~#2~argument(s). }
964 {
965     Templates-of~type~'#1'~require~#2~argument(s).\
966     You~have~tried~to~make~a~template~for~'#1'~
967     with~#3~argument(s),~which~is~not~possible:~
968     the~number~of~arguments~must~agree.
969 }
970 \msg_new:nnnn { template } { bad-number-of-arguments }
971 { Bad-number-of~arguments~for~template~type~'#1'. }
972 {
973     A~template~may~accept~between~0~and~9~arguments.\
974     You~asked~to~use~#2~arguments:~this~is~not~supported.

```

```

975 }
976 \msg_new:nnnn { template } { bad-variable }
977 { Incorrect-variable-description-`#1'. }
978 {
979   The-argument-`#1'-is-not-of-the-form \\
980   ~-`<variable>'\\
981   ~or-\\
982   ~-`global-<variable>'\\.
983   It-must-be-given-in-one-of-these-formats-to-be-used-in-a-template.
984 }
985 \msg_new:nnnn { template } { choice-not-implemented }
986 { The-choice-`#1'-has-no-implementation. }
987 {
988   Each-choice-listed-in-the-interface-for-a-template-must-
989   have-an-implementation.
990 }
991 \msg_new:nnnn { template } { choice-no-code }
992 { The-choice-`#1'-requires-implementation-details. }
993 {
994   When-creating-template-code-using-\DeclareTemplateCode,~
995   each-choice-name-must-have-an-associated-implementation.\\
996   This-should-be-given-after-a-`='-sign:-LaTeX-did-not-find-one.
997 }
998 \msg_new:nnnn { template } { choice-requires-code }
999 { The-choice-`#2'-for-key-`#1'-requires-an-implementation. }
1000 {
1001   You-should-have-put:\\
1002   \\ \ #1::~choice-`#2 = <code> ~} \\
1003   but-LaTeX-did-not-find-any-<code>.
1004 }
1005 \msg_new:nnnn { template } { duplicate-key-interface }
1006 { Key-`#1'-appears-twice-in-interface-definition-\msg_line_context:. }
1007 {
1008   Each-key-can-only-have-one-interface-declared-in-a-template.\\
1009   LaTeX-found-two-interfaces-for-`#1'.
1010 }
1011 \msg_new:nnnn { template } { keytype-requires-argument }
1012 { The-key-type-`#1'-requires-an-argument-\msg_line_context:. }
1013 {
1014   You-should-have-put:\\
1015   \\ \ <key-name>-`#1-`-<argument>-} \\
1016   but-LaTeX-did-not-find-an-<argument>.
1017 }
1018 \msg_new:nnnn { template } { invalid-keytype }
1019 { The-key-`#1'-is-missing-a-key-type-\msg_line_context:. }
1020 {
1021   Each-key-in-a-template-requires-a-key-type,~given-in-the-form:\\
1022   \\ \ <key>-`-<key-type>\\
1023   LaTeX-could-not-find-a-<key-type>-in-your-input.
1024 }
1025 \msg_new:nnnn { template } { key-no-value }
1026 { The-key-`#1'-has-no-value-\msg_line_context:. }
1027 {
1028   When-creating-an-instance-of-a-template-

```

```

1029     every-key-listed-must-include-a-value:\\
1030     \ \ <key>~::~<value>
1031 }
1032 \msg_new:nnnn { template } { key-no-variable }
1033 { The-key~'#1'~requires-implementation-details~\msg_line_context:. }
1034 {
1035     When-creating-template-code-using~\DeclareTemplateCode,~
1036     each-key-name-must-have-an-associated-implementation.\\
1037     This-should-be-given-after-a-~'='~sign:~LaTeX-did-not-find-one.
1038 }
1039 \msg_new:nnnn { template } { key-not-implemented }
1040 { Key~'#1'~has-no-implementation~\msg_line_context:. }
1041 {
1042     The-definition-of-key-implementations-for-template~'#2'~
1043     of-template-type~'#3'~does-not-include-any-details-for-key~'#1'.\\
1044     The-key-was-declared-in-the-interface-definition,~
1045     and-so-an-implementation-is-required.
1046 }
1047 \msg_new:nnnn { template } { missing-keytype }
1048 { The-key~'#1'~is-missing-a-key-type~\msg_line_context:. }
1049 {
1050     Key-interface-definitions-should-be-of-the-form\\
1051     \ \ #1~::~<key-type>\\
1052     but-LaTeX-could-not-find-a~<key-type>.
1053 }
1054 \msg_new:nnnn { template } { no-template-code }
1055 {
1056     The-template~'#2'~of-type~'#1'~is-unknown~
1057     or-has-no-implementation.
1058 }
1059 {
1060     There-is-no-code-available-for-the-template-name-given.\\
1061     This-should-be-given-using~\DeclareTemplateCode.
1062 }
1063 \msg_new:nnnn { template } { type-already-defined }
1064 { Template-type~'#1'~already-defined. }
1065 {
1066     You-have-used~\NewTemplateType~
1067     with-a-template-type-that-has-already-been-defined.
1068 }
1069 \msg_new:nnnn { template } { type-mismatch }
1070 { Template-types~'#1'~and~'#2'~do-not-agree. }
1071 {
1072     You-are-trying-to-use-a-template-directly-with~\UseInstance
1073     (or-a-similar-function),~but-the-template-types-do-not-match.
1074 }
1075 \msg_new:nnnn { template } { unknown-attribute }
1076 { The-template-attribute~'#1'~is-unknown. }
1077 {
1078     There-is-a-definition-in-the-current-template-reading\\
1079     \ \ \token_to_str:N \KeyValue {~#1~} \\
1080     but-there-is-no-key-called~'#1'.
1081 }
1082 \msg_new:nnnn { template } { unknown-choice }

```

```

1083 { The~choice~'#2'~was~not~declared~for~key~'#1'. }
1084 {
1085   The~key~'#1'~takes~a~fixed~list~of~choices~
1086   and~this~list~does~not~include~'#2'.
1087 }
1088 \msg_new:nnnn { template } { unknown-default-choice }
1089 { The~default~choice~'#2'~was~not~declared~for~key~'#1'. }
1090 {
1091   The~key~'#1'~takes~a~fixed~list~of~choices~
1092   and~this~list~does~not~include~'#2'.
1093 }
1094 \msg_new:nnnn { template } { unknown-instance }
1095 { The~instance~'#2'~of~type~'#1'~is~unknown. }
1096 {
1097   You~have~asked~to~use~an~instance~'#2',~
1098   but~this~has~not~been~created.
1099 }
1100 \msg_new:nnnn { template } { unknown-key }
1101 { Unknown~template~key~'#1'. }
1102 {
1103   The~key~'#1'~was~not~declared~in~the~interface~
1104   for~the~current~template.
1105 }
1106 \msg_new:nnnn { template } { unknown-keytype }
1107 { The~key~type~'#1'~is~unknown. }
1108 {
1109   Valid~key~types~are:\\
1110   --boolean;\\
1111   --choice;\\
1112   --commalist;\\
1113   --function;\\
1114   --instance;\\
1115   --integer;\\
1116   --length;\\
1117   --muskip;\\
1118   --real;\\
1119   --skip;\\
1120   --tokenlist.
1121 }
1122 \msg_new:nnnn { template } { unknown-type }
1123 { The~template~type~'#1'~is~unknown. }
1124 {
1125   A~template~type~needs~to~be~defined~with~\NewTemplateType
1126   prior~to~using~it.
1127 }
1128 \msg_new:nnnn { template } { unknown-template }
1129 { The~template~'#2'~of~type~'#1'~is~unknown. }
1130 {
1131   No~interface~has~been~declared~for~a~template~
1132   '#2'~of~template~type~'#1'.
1133 }

```

Information messages only have text: more text should not be needed.

```

1134 \msg_new:nnn { template } { declare-instance }
1135 { Declaring~instance~'#1'~of~type~#2~\msg_line_context:. }

```

```

1136 \msg_new:nnn { template } { declare-template-code }
1137 { Declaring~code~for~template~'#2'~of~template~type~'#1'~\msg_line_context:. }
1138 \msg_new:nnn { template } { declare-template-interface }
1139 {
1140   Declaring~interface~for~template~'#2'~of~template~type~'#1'~
1141   \msg_line_context:.
1142 }
1143 \msg_new:nnn { template } { declare-type }
1144 { Declaring~template~type~'#1'~taking~#2~argument(s)~\msg_line_context:. }
1145 \msg_new:nnn { template } { show-attribute }
1146 {
1147   The~template~'#2'~of~type~'#1'~has~
1148   \tl_if_empty:nTF {#4} { no~#3. } { #3 : #4 }
1149 }
1150 \msg_new:nnn { template } { show-values }
1151 {
1152   The~instance~'#2'~of~type~'#1'~has~
1153   \tl_if_empty:nTF {#3} { no~values. } { values: #3 }
1154 }

```

Also add template to the LaTeX messages.

```

1155 \prop_gput:Nnn \g_msg_module_type_prop { template } { LaTeX }

```

12.15 User functions

All simple translations.

```

\NewTemplateType \DeclareTemplateInterface \DeclareTemplateCode
\DeclareTemplateCopy \EditTemplateDefaults \UseTemplate
\DeclareInstance \DeclareInstanceCopy \EditInstance
\UseInstance
1156 \cs_new_protected:Npn \NewTemplateType #1#2
1157 { \__template_define_type:nn {#1} {#2} }
1158 \cs_new_protected:Npn \DeclareTemplateInterface #1#2#3#4
1159 { \__template_declare_template_keys:nnnn {#1} {#2} {#3} {#4} }
1160 \cs_new_protected:Npn \DeclareTemplateCode #1#2#3#4#5
1161 { \__template_declare_template_code:nnnnn {#1} {#2} {#3} {#4} {#5} }
1162 \cs_new_protected:Npn \DeclareTemplateCopy #1#2#3
1163 { \__template_template_set_eq:nnn {#1} {#2} {#3} }
1164 \cs_new_protected:Npn \EditTemplateDefaults #1#2#3
1165 { \__template_edit_defaults:nnn {#1} {#2} {#3} }
1166 \cs_new_protected:Npn \UseTemplate #1#2#3
1167 { \__template_use_template:nnn {#1} {#2} {#3} }
1168 \cs_new_protected:Npn \DeclareInstance #1#2#3#4
1169 { \__template_declare_instance:nnnn {#1} {#3} {#2} {#4} }
1170 \cs_new_protected:Npn \DeclareInstanceCopy #1#2#3
1171 { \__template_instance_set_eq:nnn {#1} {#2} {#3} }
1172 \cs_new_protected:Npn \EditInstance #1#2#3
1173 { \__template_edit_instance:nnn {#1} {#2} {#3} }
1174 \cs_new_protected:Npn \UseInstance #1#2
1175 { \__template_use_instance:nn {#1} {#2} }

```

(End of definition for \NewTemplateType and others. These functions are documented on page 3.)

The show functions are again just translation.

```

\ShowTemplateCode \ShowTemplateDefaults \ShowTemplateInterface
\ShowTemplateVariables \ShowInstanceValues
1176 \cs_new_protected:Npn \ShowTemplateCode #1#2
1177 { \__template_show_code:nn {#1} {#2} }
1178 \cs_new_protected:Npn \ShowTemplateDefaults #1#2
1179 { \__template_show_defaults:nn {#1} {#2} }

```

```

1180 \cs_new_protected:Npn \ShowTemplateInterface #1#2
1181   { \__template_show_keytypes:nn {#1} {#2} }
1182 \cs_new_protected:Npn \ShowTemplateVariables #1#2
1183   { \__template_show_vars:nn {#1} {#2} }
1184 \cs_new_protected:Npn \ShowInstanceValues #1#2
1185   { \__template_show_values:nn {#1} {#2} }

```

(End of definition for `\ShowTemplateCode` and others. These functions are documented on page 9.)

\IfInstanceExistsT More direct translation.

```

\IfInstanceExistsF
\IfInstanceExistsTF
1186 \cs_new:Npn \IfInstanceExistsTF #1#2
1187   { \__template_if_instance_exist:nnTF {#1} {#2} }
1188 \cs_new:Npn \IfInstanceExistsT #1#2
1189   { \__template_if_instance_exist:nnT {#1} {#2} }
1190 \cs_new:Npn \IfInstanceExistsF #1#2
1191   { \__template_if_instance_exist:nnF {#1} {#2} }

```

(End of definition for `\IfInstanceExistsT`, `\IfInstanceExistsF`, and `\IfInstanceExistsTF`. These functions are documented on page 7.)

\KeyValue Simply dump the argument when executed: this should not happen.

```

1192 \cs_new_protected:Npn \KeyValue #1 {#1}

```

(End of definition for `\KeyValue`. This function is documented on page 4.)

\AssignTemplateKeys A short call to use a token register by proxy.

```

1193 \cs_new_protected:Npn \AssignTemplateKeys { \__template_assignments_pop: }

```

(End of definition for `\AssignTemplateKeys`. This function is documented on page 5.)

\SetTemplateKeys A friendly wrapper

```

1194 \cs_new_protected:Npn \SetTemplateKeys #1#2#3
1195   { \keys_set_known:nnN { template / #1 / #2 } {#3} \l__template_tmp_clist }

```

(End of definition for `\SetTemplateKeys`. This function is documented on page 9.)

```

1196 <latexrelease>\IncludeInRelease{0000/00/00}{lttemplates}%
1197 <latexrelease>           {Prototype-document-commands}%
1198 <latexrelease>
1199 <latexrelease>\EndModuleRelease

```

```

1200 \ExplSyntaxOff
1201 </2ekernel | latexrelease>

```

We need to stop DocStrip treating @@ in a special way at this point.

```

1202 <@@=)

```


Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
<code>\</code>	965, 973, 979, 980, 981, 982, 995, 1001, 1002, 1008, 1014, 1015, 1021, 1022, 1029, 1036, 1043, 1050, 1051, 1060, 1078, 1079, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1116, 1117, 1118, 1119
<code>_</code>	1002, 1015, 1022, 1030, 1051, 1079
A	
<code>\AssignTemplateKeys</code>	<u>1193</u> , 387, 5
B	
bool commands:	
<code>\bool_if:NTF</code>	222, 228, 658, 761, 810, 827, 849
<code>\bool_new:N</code>	21, 22
<code>\bool_set_false:N</code>	277, 656, 737
<code>\bool_set_true:N</code> ...	248, 290, 615, 746
<code>\l_tmpa_bool</code>	5
C	
<code>\c</code>	383
<code>\caption</code>	<u>2</u>
clist commands:	
<code>\clist_if_empty:NTF</code>	534, 563
<code>\clist_if_in:NnTF</code>	266, 570
<code>\clist_map_inline:Nn</code>	536
<code>\clist_new:N</code>	32
<code>\clist_remove_all:Nn</code>	572
<code>\clist_set:Nn</code>	524
<code>\l_tmpa_clist</code>	5
cs commands:	
<code>\cs_generate_from_arg_count:NNnn</code>	397, 459, 818
<code>\cs_generate_variant:Nn</code>	64, 349, 502, 703, 734, 807
<code>\cs_gset:Npn</code>	811
<code>\cs_gset_eq:NN</code>	642
<code>\cs_gset_protected:Npn</code>	399, 828
<code>\cs_if_exist:NTF</code>	54, 60, 73, 86, 395, 453, 488, 640, 663, 679
<code>\cs_if_exist_use:NTF</code>	728
<code>\cs_new:Npn</code>	908, 1186, 1188, 1190, 350, 351, 454, 508
<code>\cs_new_eq:NN</code>	356, 357, 358, 373
<code>\cs_new_protected:Npe</code>	275
<code>\cs_new_protected:Npn</code>	872, 874, 885, 890, 896, 902, 909, 911, 913, 922, 931, 940, 948, 42, 1156, 1158, 1160, 1162, 1164, 1166, 1168, 1170, 1172, 1174, 1176, 1178, 1180, 1182, 1184, 1192, 1193, 1194, 52, 58, 65, 71, 96, 104, 120, 128, 136, 146, 162, 172, 182, 188, 203, 219, 241, 253, 270, 296, 320, 352, 354, 359, 361, 363, 365, 367, 369, 371, 374, 393, 401, 412, 414, 428, 432, 446, 503, 522, 540, 561, 578, 585, 593, 598, 607, 613, 618, 627, 632, 645, 654, 673, 686, 698, 704, 710, 715, 735, 743, 748, 759, 765, 785, 802, 808, 814, 825, 831, 843, 853
D	
debug commands:	
<code>\debug_resume:</code>	102, 118, 126, 134
<code>\debug_suspend:</code>	98, 106, 122, 130
<code>\DeclareInstance</code>	<u>1156</u> , 7
<code>\DeclareInstanceCopy</code>	<u>1156</u> , 7
<code>\DeclareTemplateCode</code>	994, 1035, 1061, <u>1156</u> , 3
<code>\DeclareTemplateCopy</code>	<u>1156</u> , 6
<code>\DeclareTemplateInterface</code>	<u>1156</u> , 3
dim commands:	
<code>\dim_eval:n</code>	364
<code>\dim_new:N</code>	33
<code>\l_tmpa_dim</code>	5
E	
<code>\EditInstance</code>	<u>1156</u> , 8
<code>\EditTemplateDefaults</code>	<u>1156</u> , 8
<code>\EndModuleRelease</code>	1199
exp commands:	
<code>\exp_after:wN</code>	873, 880, 285, 740
<code>\exp_args:Ne</code>	845
<code>\exp_not:N</code>	277, 278, 279, 280, 290, 296, 299, 301, 302, 306, 308, 311, 316, 460, 461, 476, 478, 494, 667, 669, 772, 780, 820, 835, 860, 867
<code>\exp_not:n</code>	282, 463, 588, 668, 773, 774, 781, 819, 821, 822, 835, 838, 839, 860, 861, 867, 868

\ExplSyntaxOff	1200	1047, 1054, 1063, 1069, 1075, 1082,
\ExplSyntaxOn	6	1088, 1094, 1100, 1106, 1122, 1128
F		
fp commands:		
\fp_eval:n	368	\msg_show:nnnnn 953
\l_tmpa_fp	5	\msg_show:nnnnnn 942
		\msg_show_item_unbraced:nn 946, 958
		muskip commands:
		\muskip_eval:n 366
		\muskip_new:N 35
		\l_tmpa_muskip 5
I		
\IfInstanceExistsF	1186, 7	
\IfInstanceExistsT	1186, 7	
\IfInstanceExistsTF	1186, 7	
\IncludeInRelease	1196	
int commands:		
\int_compare:nNnTF	45	
\int_compare:nTF	191	
\int_eval:n	362	
\int_new:N	34	
\int_set:Nn	190	
\l_tmpa_int	5	
K		
kernel internal commands:		
_kernel_quark_new_conditional:Nn	41	
keys commands:		
\keys_define:nn	505, 526, 587	
\keys_set_known:nnN	1195	
keyval commands:		
\keyval_parse:NNn	212, 610	
\keyval_parse:nnn	406, 527	
\KeyValue	874, 1079, 1192, 79, 4	
M		
\message	3	
msg commands:		
\msg_error:nn	267	
\msg_error:nnn	883,	
62, 69, 185, 233, 247, 291, 313,		
413, 426, 443, 499, 537, 616, 625, 731		
\msg_error:nnnn	900, 906,	
56, 75, 199, 555, 583, 595, 684, 696, 796		
\msg_error:nnnnn	48, 410	
\msg_info:nnnn		
109, 193, 396, 641, 664, 680		
\msg_line_context:		
1006, 1012, 1019, 1026, 1033,		
1040, 1048, 1135, 1137, 1141, 1144		
\g_msg_module_type_prop	1155	
\msg_new:nnn		
1134, 1136, 1138, 1143, 1145, 1150		
\msg_new:nnnn		
962, 970, 976, 985, 991, 998,		
1005, 1011, 1018, 1025, 1032, 1039,		
1047, 1054, 1063, 1069, 1075, 1082,		
1088, 1094, 1100, 1106, 1122, 1128		
N		
\NewModuleRelease	7	
\NewTemplateType	1066, 1125, 1156, 3	
P		
prg commands:		
\prg_generate_conditional_-		
variant:Nnn	83	
\prg_new_conditional:Npnn	77, 84, 90	
\prg_return_false:	81, 88, 94	
\prg_return_true:	80, 87, 93	
prop commands:		
\prop_clear:N		
144, 154, 170, 180, 209, 210, 405		
\prop_clear_new:N	123	
\prop_gclear:N	110	
\prop_gclear_new:N	99, 131	
\prop_get:NnN		
44, 550, 553, 580, 691, 712, 793		
\prop_get:NnNTF		
877, 418, 531, 622, 717, 719, 804		
\prop_gput:Nnn	1155, 195	
\prop_gset_eq:NN	100, 113, 132	
\prop_if_exist:NTF		
107, 138, 148, 164, 174		
\prop_if_in:NnTF	67, 184, 544, 548	
\prop_map_function:NN	946, 957	
\prop_map_inline:Nn	409	
\prop_new:N	28, 30, 31, 112, 18	
\prop_put:Nnn	261, 353,	
355, 360, 467, 481, 496, 525, 591, 660		
\prop_remove:Nn	424	
\prop_set_eq:NN	124, 141, 151, 167, 177	
Q		
quark commands:		
\q_nil	79, 92	
\quark_new:N	40	
\q_stop	79, 92	
quark internal commands:		
\q_template_nil	881, 40, 12	
R		
regex commands:		
\regex_match:nnTF	383	

S

scan commands:
 \scan_new:N 38, 39

scan internal commands:
 \s_template_mark 881, 38, 12
 \s_template_stop
 881, 885, 39, 286, 297, 308,
 329, 343, 351, 430, 433, 740, 743, 12

\section 2

seq commands:
 \seq_clear:N 160, 211
 \seq_const_from_clist:Nn 16
 \seq_gclear_new:N 115
 \seq_gset_eq:NN 116
 \seq_if_exist:NnTF 155
 \seq_if_in:NnTF 230
 \seq_map_break: 249, 340
 \seq_map_function:NN .. 226, 346, 707
 \seq_new:N 29
 \seq_put_right:Nn 263
 \seq_set_eq:NN 157

\SetTemplateKeys 1194, 9
\ShowInstanceValues 1176, 9
\ShowTemplateCode 1176, 9
\ShowTemplateDefaults 1176, 9
\ShowTemplateInterface 1176, 9
\ShowTemplateVariables 1176, 10

skip commands:
 \skip_eval:n 370
 \skip_new:N 36
 \l_tmpa_skip 5

str commands:
 \str_case:nn 510
 \str_case:nnTF 448
 \str_if_eq:nnTF 898, 79, 92,
 243, 264, 462, 477, 493, 565, 722, 724

T

template internal commands:
 __template_assign_boolean: 759, 759
 __template_assign_boolean_aux:n
 759, 762, 763, 765
 __template_assign_choice: . 785, 785
 __template_assign_choice_-
 aux:nTF 785, 787, 790, 802, 807
 __template_assign_function: 808, 808
 __template_assign_function_-
 aux:N 808, 811, 812, 814
 __template_assign_instance: 825, 825
 __template_assign_instance_-
 aux:N 825, 828, 829, 831
 __template_assign_variable: ...
 729, 843, 843

__template_assign_variable:n ...
 843, 845, 853

__template_assignments_pop: ...
 908, 908, 1193

__template_assignments_push:n ..
 909, 909, 667

\l__template_assignments_tl
 908, 910, 668, 706, 770,
 778, 19, 805, 816, 833, 858, 865, 11

\c__template_code_root_tl 912, 54,
 395, 398, 640, 642, 643, 669, 9, 756, 10

__template_convert_to_assignments:
 662, 704, 704, 755

__template_convert_to_assignments_-
 aux:n 704, 708, 710

__template_convert_to_assignments_-
 aux:nn 704, 713, 715, 734

__template_declare_instance:nnnn
 1169, 645, 645

__template_declare_instance_-
 aux:nnnn 645, 651, 654, 701

__template_declare_template_-
 code:nnnn 374, 384, 386, 393

__template_declare_template_-
 code:nnnnn 1161, 374, 374

__template_declare_template_-
 keys:nnnn 1159, 203, 203

__template_declare_type:nn
 182, 186, 188

\l__template_default_tl 20, 11

\c__template_defaults_root_tl ...
 99, 100, 139, 142, 10, 10

__template_define_type:nn
 1157, 182, 182

__template_edit_defaults:nnn ...
 1165, 598, 598

__template_edit_instance:nnn ...
 1173, 686, 686

__template_edit_instance_-
 aux:nnnn 693, 698, 703

__template_edit_instance_-
 aux:nnnnn 686

\l__template_error_bool . 222, 228,
 248, 277, 290, 615, 656, 658, 21, 11

__template_execute_if_arg_-
 agree:nnTF 42, 42, 207, 378

__template_execute_if_code_-
 exist:nnTF .. 933, 52, 52, 647, 750

__template_execute_if_keys_-
 exist:nnTF 71

__template_execute_if_keytype_-
 exist:nTF 58, 58, 64, 224

__template_execute_if_type_-
 exist:nTF 65, 65, 205, 376

```

__template_find_global: 725, 735, 735
__template_find_global_aux:w ...
    ..... 735, 740, 743
l__template_global_bool .....
    .. 737, 746, 761, 810, 827, 849, 22, 11
__template_if_instance_exist:nn 84
__template_if_instance_exist:nnTF
    904, 950, 1187, 1189, 1191, 84, 675, 688
__template_if_key_value:n .. 77, 83
__template_if_key_value:nTF ...
    ..... 77, 767, 855
__template_if_keys_exist:nnTF ..
    ..... 915, 924, 71, 380, 600
__template_if_use_template:n ... 90
__template_if_use_template:nTF .
    ..... 892, 90
__template_implement_choice_-
    elt:n ..... 528, 561, 593
__template_implement_choice_-
    elt:nnn ..... 529, 561, 561
__template_implement_choice_-
    elt_aux:n ..... 561, 567, 575, 578
__template_implement_choice_-
    elt_aux:nnn ..... 561, 566, 573, 585
__template_implement_choices:nn
    ..... 450, 522, 522
__template_implement_choices_-
    default: ..... 522, 533, 540
__template_instance_set_eq:nnn .
    ..... 1171, 673, 673
c__template_instances_root_tl ..
    905, 86, 663, 665, 679, 681, 682, 11, 10
l__template_key_name_tl .....
    .... 231, 234, 261, 263, 284, 299,
    306, 311, 353, 355, 360, 416, 419,
    424, 468, 482, 497, 506, 525, 526,
    531, 543, 547, 550, 553, 556, 557,
    580, 583, 588, 590, 596, 620, 622,
    625, 727, 788, 791, 793, 797, 23, 26
c__template_key_order_root_tl ..
    ..... 115, 116, 155, 158, 13, 10
l__template_key_order_seq .. 29,
    117, 157, 160, 211, 230, 263, 707, 11
__template_key_to_value: .....
    ..... 872, 872, 769, 857
__template_key_to_value_auxi:w .
    ..... 872, 873, 874
__template_key_to_value_auxii:w
    ..... 872, 880, 885
l__template_keytype_arg_tl ....
    ..... 25, 245, 258, 259,
    266, 323, 337, 463, 524, 821, 838, 11
l__template_keytype_tl . 24, 224,
    243, 257, 264, 273, 322, 333, 420,
    422, 448, 510, 630, 722, 724, 728, 11
c__template_keytypes_arg_seq ...
    ..... 226, 346, 16, 11
l__template_keytypes_prop .. 28,
    927, 114, 151, 154, 210, 261, 409,
    418, 424, 550, 580, 622, 712, 793, 11
c__template_keytypes_root_tl 73,
    107, 110, 112, 113, 149, 152, 12, 10
__template_map_var_type: .....
    ..... 489, 492, 508, 508, 847
__template_parse_keys_elt:n ...
    ..... 213, 219, 219, 272, 18
__template_parse_keys_elt:nn ...
    ..... 213, 270, 270
__template_parse_keys_elt_aux: .
    ..... 219, 236, 253
__template_parse_keys_elt_aux:n
    ..... 219, 227, 241
__template_parse_values:nnn ...
    ..... 603, 607, 607, 657, 754
__template_parse_values_elt:n ..
    ..... 611, 613, 613
__template_parse_values_elt:nn .
    ..... 611, 618, 618
__template_parse_values_elt_-
    aux:n ..... 618, 624, 627
__template_parse_vars_elt:n ...
    ..... 407, 412, 412
__template_parse_vars_elt:nnn ..
    ..... 407, 414, 414
__template_parse_vars_elt_-
    aux:nn ..... 414, 423, 428
__template_parse_vars_elt_-
    aux:nnn ..... 414, 436, 440, 446, 502
__template_parse_vars_elt_-
    aux:nw ..... 414, 430, 432
__template_parse_vars_elt_-
    key:nn ..... 414, 455, 472, 490, 503
__template_quark_if_nil:N ..... 41
__template_quark_if_nil:NTF .. 887
__template_quark_if_nil:nTF ... 41
__template_quark_if_nil_p:n ... 41
__template_recover_defaults:nn .
    917, 136, 136, 403, 602, 634, 649, 752
__template_recover_keytypes:nn .
    ..... 926, 136, 146, 404, 609, 636
__template_recover_values:nn ...
    ..... 952, 136, 162, 677, 690
__template_recover_vars:nn ....
    .... 935, 136, 172, 638, 650, 700, 753
c__template_restrict_root_tl ... 10

```

<code>__template_show:Nnnn</code>	<code>__template_store_value_tokenlist:n</code>
..... 913 , 918 , 927 , 936 , 940 359 , 371 , 373
<code>__template_show_code:nn</code>	<code>__template_store_values:nn</code>
..... 911 , 911 , 1177 96 , 120 , 661 , 678
<code>__template_show_defaults:nn</code> ...	<code>__template_store_vars:nn</code>
..... 913 , 913 , 1179 96 , 128 , 408 , 639
<code>__template_show_keytypes:nn</code> ...	<code>__template_template_set_eq:nnn</code> .
..... 913 , 922 , 1181 1163 , 632 , 632
<code>__template_show_values:nn</code>	<code>\l__template_tmp_clist</code>
..... 948 , 948 , 1185	1195 , 524 , 534 , 536 , 563 , 570 , 572 , 12
<code>__template_show_vars:nn</code>	<code>\l__template_tmp_dim</code>
..... 913 , 931 , 1183	33 , 12
<code>__template_split_keytype:n</code>	<code>\l__template_tmp_int</code>
..... 221 , 275 , 275 34 , 190 , 191 , 194 , 196 , 200 , 12
<code>__template_split_keytype_arg:n</code> .	<code>\l__template_tmp_muskip</code>
..... 316 , 320 ,	35 , 12
320 , 349 , 422 , 552 , 582 , 629 , 721 , 795	<code>\l__template_tmp_skip</code>
<code>__template_split_keytype_arg_-</code>	36 , 12
<code>aux:n</code>	<code>\l__template_tmp_tl</code>
320 , 324 , 347 , 350	876 ,
<code>__template_split_keytype_arg_-</code>	877 , 883 , 37 , 44 , 45 , 49 , 255 , 262 ,
<code>aux:w</code>	278 , 279 , 280 , 286 , 532 , 542 , 543 ,
320 , 328 , 343 , 351	544 , 546 , 547 , 548 , 551 , 552 , 554 ,
<code>__template_split_keytype_aux:w</code> .	581 , 582 , 589 , 591 , 623 , 629 , 692 ,
.....	694 , 712 , 713 , 794 , 795 , 804 , 805 , 12
275 , 285 , 296 , 308	<code>\g__template_type_prop</code>
<code>__template_store_defaults:nn</code> 44 , 67 , 184 , 195 , 18 , 11
..... 96 , 96 , 214 , 604 , 635	<code>__template_use_instance:nn</code>
<code>__template_store_key_implementation:nnn</code> 890 , 890 , 1175 , 837
.....	<code>__template_use_instance_aux:nn</code> .
382 , 401 , 401 890 , 894 , 902
<code>__template_store_keytypes:nn</code> ...	<code>__template_use_instance_-</code>
..... 96 , 104 , 215 , 637	<code>aux:nNnnn</code>
<code>__template_store_value:n</code>	890 , 893 , 896
..... 354 , 354 , 356 , 357 , 358	<code>__template_use_template:nnn</code> ...
<code>__template_store_value_aux:Nn</code> 899 , 1167 , 748 , 748
359 , 359 , 362 , 364 , 366 , 368 , 370 , 372	<code>\l__template_value_tl</code> ..
<code>__template_store_value_boolean:n</code>	873 , 878 ,
.....	880 , 888 , 26 , 717 , 767 , 774 , 780 ,
352 , 352	788 , 798 , 822 , 839 , 855 , 861 , 868 , 11
<code>__template_store_value_choice:n</code>	<code>\l__template_values_prop</code>
..... 918 , 30 , 957 , 101 ,
354 , 356	125 , 141 , 144 , 167 , 170 , 209 , 353 ,
<code>__template_store_value_commalist:n</code>	355 , 360 , 531 , 553 , 660 , 691 , 717 , 20
.....	<code>\c__template_values_root_tl</code>
359 , 373 123 , 124 , 165 , 168 , 14 , 10
<code>__template_store_value_function:n</code>	<code>\l__template_var_tl</code>
..... 27 , 719 , 738 , 740 ,
354 , 357	745 , 773 , 781 , 819 , 835 , 860 , 867 , 11
<code>__template_store_value_instance:n</code>	<code>\l__template_vars_prop</code> ..
.....	877 , 936 ,
354 , 358	31 , 133 , 177 , 180 , 405 , 467 , 481 ,
<code>__template_store_value_integer:n</code>	496 , 525 , 544 , 548 , 591 , 719 , 804 , 11
.....	<code>\c__template_vars_root_tl</code>
359 , 361 131 , 132 , 175 , 178 , 15 , 10
<code>__template_store_value_length:n</code>	tl commands:
.....	<code>\c_space_tl</code> ...
359 , 363	543 , 547 , 590 , 788 , 791
<code>__template_store_value_muskip:n</code>	<code>\tl_clear:N</code>
.....	284 , 323 , 706
359 , 365	<code>\tl_const:Nn</code> ..
<code>__template_store_value_real:n</code> ..	9 , 10 , 11 , 12 , 13 , 14 , 15
.....	<code>\tl_head:w</code>
359 , 367	79 , 92
<code>__template_store_value_skip:n</code> ..	<code>\tl_if_blank:nTF</code> ...
.....	331 , 335 , 435 , 438
359 , 369	

\tl_if_empty:NTF	245, 258, 311	\tl_trim_spaces:n	876, 301, 322, 334, 417, 441, 621
\tl_if_empty:nTF	1148, 1153	\l_tmpa_tl	5
\tl_if_in:nnTF	280, 304, 326, 738	token commands:	
\tl_if_single:nTF	486	\token_to_str:N	1079, 279, 280, 297, 304, 307, 314
\tl_new:N .	24, 25, 26, 27, 37, 19, 20, 23		
\tl_put_right:Nn	299, 306, 770, 778, 805, 816, 833, 858, 865		
\tl_replace_all:Nnn	279		
\tl_set:Nn	876, 888, 910, 255, 278, 322, 333, 337, 416, 542, 546, 589, 620, 727, 745	U	
\tl_to_str:n	876, 943, 944, 945, 954, 955, 302, 417, 621	use commands:	
		\use:N	905, 273, 489, 630, 756
		\use:n	294, 338, 372
		\use_i:mn	5
		\UseInstance	1072, <u>1156</u> , 478, 8
		\UseTemplate	<u>1156</u> , 92, 8