The xtemplate package
Prototype document functions[*]

The LaTeX3 Project[†]

Released 2012/09/05

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 TeX 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 TeX implementation in the middle is the glue between the two.

LaTeX’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 LaTeX 2ε classes look somewhat dated now in terms of their visual design, their typography is generally sound. (Barring the occasional minor faults.)

However, LaTeX 2ε 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 xtemplate 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. xtemplate also makes it easier for LaTeX programmers to provide their own customisations on top of a pre-existing class.

[*]This file describes v4205, last revised 2012/09/05.
[†]E-mail: latex-team@latex-project.org

1 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 object types, templates, and instances, and they are discussed below in sections 3, 4, and 6, respectively.

2 Objects, templates, and instances

By formally declaring documents to be composed of mark-up elements grouped into objects, 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.

3 Object types

An *object type* (sometimes just "object") 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 object, for example, might take three inputs: "title", "short title", and "label".

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

`\DeclareObjectType`

`\DeclareObjectType {⟨object type⟩} {⟨no. of args⟩}`

This function defines an ⟨*object type*⟩ taking ⟨*number of arguments*⟩, where the ⟨*object type*⟩ is an abstraction as discussed above. For example,

```
\DeclareObjectType{sectioning}{3}
```

creates an object type "sectioning", where each use of that object type will need three arguments.

4 Templates

A *template* is a generalised design solution for representing the information of a specified object type. Templates that do the same thing, but in different ways, are grouped together by their object 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`

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

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

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.

Key-type	Description of input
`boolean`	`true` or `false`
`choice{`⟨*choices*⟩`}`	A list of pre-defined ⟨*choices*⟩
`code`	Generalised key type: use `#1` as the input to the key
`commalist`	A comma-separated list
`function{`⟨N⟩`}`	A function definition with N arguments (N from 0 to 9)
`instance{`⟨*name*⟩`}`	An instance of type ⟨*name*⟩
`integer`	An integer or integer expression
`length`	A fixed length
`muskip`	A math length with shrink and stretch components
`real`	A real (floating point) value
`skip`	A length with shrink and stretch components
`tokenlist`	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 { object } { 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`	Boolean variable, *e.g.* `\l_tmpa_bool`
`choice`	List of choice implementations (see Section 5)
`code`	⟨*code*⟩ using `#1` as input to the key
`commalist`	Comma list, *e.g.* `\l_tmpa_clist`
`function`	Function taking N arguments, *e.g.* `\use_i:nn`
`instance`	
`integer`	Integer variable, *e.g.* `\l_tmpa_int`
`length`	Dimension variable, *e.g.* `\l_tmpa_dim`
`muskip`	Muskip variable, *e.g.* `\l_tmpa_muskip`
`real`	Floating-point variable, *e.g.* `\l_tmpa_fp`
`skip`	Skip variable, *e.g.* `\l_tmpa_skip`
`tokenlist`	Token list variable, *e.g.* `\l_tmpa_tl`

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`

```
⟨key1⟩ = ⟨variable1⟩,
⟨key2⟩ = ⟨variable2⟩,
⟨key3⟩ = global ⟨variable3⟩,
⟨key4⟩ = global ⟨variable4⟩,
...
```

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 ⟨*object 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*⟩.

With the exception of the choice, code and function key types, the ⟨*variable*⟩ here should be the name of an existing LaTeX3 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 object type.

`\AssignTemplateKeys`

```
\AssignTemplateKeys
```

In the final argument of `\DeclareTemplateCode` the assignment of keys defined by the template is carried out by using the function `\AssignTemplateKeys`. Thus no keys are assigned if this is missing from the ⟨*code*⟩ used.

`\EvaluateNow` `\EvaluteNow {⟨expression⟩}`

The standard method when creating an instance from a template is to evaluate the ⟨*expression*⟩ when the instance is used. However, it may be desirable to calculate the value when declared, which can be forced using `\EvaluateNow`. Currently, this functionality is regarded as experimental: the team have not found an example where it is actually needed, and so it may be dropped *if* no good examples are suggested!

5 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.

6 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
  {⟨object type⟩} {⟨instance⟩} {⟨template⟩} {⟨parameters⟩}
```

This function uses a ⟨*template*⟩ for an ⟨*object type*⟩ to create an ⟨*instance*⟩. The ⟨*instance*⟩ will be set up using the ⟨*parameters*⟩, which will set some of the ⟨*keys*⟩ in the ⟨*template*⟩.

As a practical example, consider an object type for document sections (which might include chapters, parts, sections, *etc.*), which is called `sectioning`. One possible template for this object 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.

7 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`
`{⟨object type⟩} {⟨instance⟩} ⟨arguments⟩`

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

`\UseTemplate`

`\UseTemplate {⟨object type⟩} {⟨template⟩}`
`{⟨settings⟩} ⟨arguments⟩`

Uses the ⟨*template*⟩ of the specified ⟨*object type*⟩, applying the ⟨*settings*⟩ and absorbing ⟨*arguments*⟩ as detailed by the ⟨*object 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 { object } { template } { instance }
  {
    instance-key =
      \UseTemplate { object2 } { template2 } { <settings> }
  }
```

8 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`
`{⟨object type⟩} {⟨template⟩} {⟨new defaults⟩}`

Edits the ⟨*defaults*⟩ for a ⟨*template*⟩ for an ⟨*object 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`
`{⟨object type⟩} {⟨instance⟩} {⟨new values⟩}`

Edits the ⟨*values*⟩ for an ⟨*instance*⟩ for an ⟨*object 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.

9 When template parameters should be frozen

A class designer may be inheriting templates declared by someone else, either third-party code or the LaTeX kernel itself. Sometimes these templates will be overly general for the purposes of the document. The user should be able to customise parts of the template instances, but otherwise be restricted to only those parameters allowed by the designer.

`\DeclareRestrictedTemplate`

```
\DeclareRestrictedTemplate
  {⟨object type⟩} {⟨parent template⟩} {⟨new template⟩}
  {⟨parameters⟩}
```

Creates a copy of the ⟨*parent template*⟩ for the ⟨*object type*⟩ called ⟨*new template*⟩. The key–value list of ⟨*parameters*⟩ applies in the ⟨*new template*⟩ and cannot be changed when creating an instance.

10 Getting information about templates and instances

`\ShowInstanceValues`

```
\ShowInstanceValues {⟨object type⟩} {⟨instance⟩}
```

Shows the ⟨*values*⟩ for an ⟨*instance*⟩ of the given ⟨*object type*⟩ at the terminal.

`\ShowCollectionInstanceValues`

```
\ShowInstanceValues {⟨collection⟩} {⟨object type⟩} {⟨instance⟩}
```

Shows the ⟨*values*⟩ for an ⟨*instance*⟩ within a ⟨*collection*⟩ of the given ⟨*object type*⟩ at the terminal.

`\ShowTemplateCode`

```
\ShowTemplateCode {⟨object type⟩} {⟨template⟩}
```

Shows the ⟨*code*⟩ of a ⟨*template*⟩ for an ⟨*object type*⟩ in the terminal.

`\ShowTemplateDefaults`

```
\ShowTemplateDefaults {⟨object type⟩} {⟨template⟩}
```

Shows the ⟨*default*⟩ values of a ⟨*template*⟩ for an ⟨*object type*⟩ in the terminal.

`\ShowTemplateInterface`

```
\ShowTemplateInterface {⟨object type⟩} {⟨template⟩}
```

Shows the ⟨*keys*⟩ and associated ⟨*key types*⟩ of a ⟨*template*⟩ for an ⟨*object type*⟩ in the terminal.

\ShowTemplateVariables

```
\ShowTemplateVariables {⟨object type⟩} {⟨template⟩}
```

Shows the ⟨*variables*⟩ and associated ⟨*keys*⟩ of a ⟨*template*⟩ for an ⟨*object 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 object 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.

11 xtemplate Implementation

```
⟨*package⟩
⟨@@=xtemplate⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

11.1 Variables and constants

\c__xtemplate_code_root_tl
\c__xtemplate_defaults_root_tl
\c__xtemplate_instances_root_tl
\c__xtemplate_keytypes_root_tl
\c__xtemplate_key_order_root_tl
\c__xtemplate_restrict_root_tl
\c__xtemplate_values_root_tl
\c__xtemplate_vars_root_tl

So that literal values are kept to a minimum.

```
\tl_const:Nn \c__xtemplate_code_root_tl      { template~code~>~ }
\tl_const:Nn \c__xtemplate_defaults_root_tl  { template~defaults~>~ }
\tl_const:Nn \c__xtemplate_instances_root_tl { template~instance~>~  }
\tl_const:Nn \c__xtemplate_keytypes_root_tl  { template~key~types~>~ }
\tl_const:Nn \c__xtemplate_key_order_root_tl { template~key~order~>~ }
\tl_const:Nn \c__xtemplate_restrict_root_tl  { template~restrictions~>~ }
\tl_const:Nn \c__xtemplate_values_root_tl    { template~values~>~ }
\tl_const:Nn \c__xtemplate_vars_root_tl      { template~vars~>~ }
```

(*End definition for* `\c__xtemplate_code_root_tl` *This function is documented on page* **??**.)

\c__xtemplate_keytypes_arg_seq

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

```
\seq_new:N \c__xtemplate_keytypes_arg_seq
\seq_put_right:Nn \c__xtemplate_keytypes_arg_seq { choice }
\seq_put_right:Nn \c__xtemplate_keytypes_arg_seq { function }
\seq_put_right:Nn \c__xtemplate_keytypes_arg_seq { instance }
```

(*End definition for* `\c__xtemplate_keytypes_arg_seq` *This variable is documented on page* **??**.)

\g__xtemplate_object_type_prop

For storing types and the associated number of arguments.

```
\prop_new:N \g__xtemplate_object_type_prop
```

(*End definition for* `\g__xtemplate_object_type_prop` *This variable is documented on page* **??**.)

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

```
\tl_new:N \l__xtemplate_assignments_tl
```

(*End definition for* `\l__xtemplate_assignments_tl` *This variable is documented on page* **??**.)

`\l__xtemplate_collection_tl` The current instance collection name is stored here.

```
\tl_new:N \l__xtemplate_collection_tl
```

(*End definition for* `\l__xtemplate_collection_tl` *This variable is documented on page* **??**.)

`\l__xtemplate_collections_prop` Lists current collection in force, indexed by object type.

```
\prop_new:N \l__xtemplate_collections_prop
```

(*End definition for* `\l__xtemplate_collections_prop` *This variable is documented on page* **??**.)

`\l__xtemplate_default_tl` The default value for a key is recovered here from the property list in which it is stored. The internal implementation of property lists means that this is safe even with un-escaped `#` tokens.

```
\tl_new:N \l__xtemplate_default_tl
```

endmacro

`\l__xtemplate_error_bool` A flag for errors to be carried forward.

```
\bool_new:N \l__xtemplate_error_bool
```

`\l__xtemplate_global_bool` Used to indicate that assignments should be global.

```
\bool_new:N \l__xtemplate_global_bool
```

`\l__xtemplate_restrict_bool` A flag to indicate that a template is being restricted.

```
\bool_new:N \l__xtemplate_restrict_bool
```

`\l__xtemplate_restrict_clist` A scratch list for restricting templates.

```
\clist_new:N \l__xtemplate_restrict_clist
```

`\l__xtemplate_key_name_tl`
`\l__xtemplate_keytype_tl`
`\l__xtemplate_keytype_arg_tl`
`\l__xtemplate_value_tl`
`\l__xtemplate_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.

```
\tl_new:N \l__xtemplate_key_name_tl
\tl_new:N \l__xtemplate_keytype_tl
\tl_new:N \l__xtemplate_keytype_arg_tl
\tl_new:N \l__xtemplate_value_tl
\tl_new:N \l__xtemplate_var_tl
```

`\l__xtemplate_keytypes_prop`
`\l__xtemplate_key_order_seq`
`\l__xtemplate_values_prop`
`\l__xtemplate_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

```
\prop_new:N \l__xtemplate_keytypes_prop
\seq_new:N \l__xtemplate_key_order_seq
\prop_new:N \l__xtemplate_values_prop
\prop_new:N \l__xtemplate_vars_prop
```

`\l__xtemplate_tmp_clist` `\l__xtemplate_tmp_dim` `\l__xtemplate_tmp_int` `\l__xtemplate_tmp_muskip` `\l__xtemplate_tmp_skip`

For pre-processing the data stored by xtemplate, a number of scratch variables are needed. The assignments are made to these in the first instance, unless evaluation is delayed.

```
\clist_new:N \l__xtemplate_tmp_clist
\dim_new:N \l__xtemplate_tmp_dim
\int_new:N \l__xtemplate_tmp_int
\muskip_new:N \l__xtemplate_tmp_muskip
\skip_new:N \l__xtemplate_tmp_skip
```

`\l__xtemplate_tmp_tl`

A scratch variable for comparisons and so on.

```
\tl_new:N \l__xtemplate_tmp_tl
```

11.2 Variant of prop functions

`\prop_get:NoNTF`

In some cases, we need to expand the key, and get the corresponding value in a property list if it exists.

```
\cs_generate_variant:Nn \prop_get:NnNTF { No }
\cs_generate_variant:Nn \prop_get:NnNT  { No }
\cs_generate_variant:Nn \prop_get:NnNF  { No }
```

11.3 Testing existence and validity

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

`\__xtemplate_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

```
\cs_new_protected:Npn \__xtemplate_execute_if_arg_agree:nnT #1#2#3
  {
    \prop_get:NnN \g__xtemplate_object_type_prop {#1} \l__xtemplate_tmp_tl
    \int_compare:nNnTF {#2} = \l__xtemplate_tmp_tl
      {#3}
      {
        \msg_error:nnxxx { xtemplate }
          { argument-number-mismatch } {#1} { \l__xtemplate_tmp_tl } {#2}
      }
  }
```

`\__xtemplate_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.

```
\cs_new_protected:Npn \__xtemplate_execute_if_code_exist:nnT #1#2#3
  {
    \cs_if_exist:cTF { \c__xtemplate_code_root_tl #1 / #2 }
     {#3}
     {
       \msg_error:nnxx { xtemplate } { no-template-code }
         {#1} {#2}
```

```
      }
  }
```

__xtemplate_execute_if_keytype_exist:nT
__xtemplate_execute_if_keytype_exist:oT

The test for valid keytypes looks for a function to set up the key, which is part of the "code" side of the template definition. This avoids having different lists for the two parts of the process.

```
\cs_new_protected:Npn \__xtemplate_execute_if_keytype_exist:nT #1#2
  {
    \cs_if_exist:cTF { __xtemplate_store_value_ #1 :n }
      {#2}
      { \msg_error:nnx { xtemplate } { unknown-keytype } {#1} }
  }
\cs_generate_variant:Nn \__xtemplate_execute_if_keytype_exist:nT { o }
```

__xtemplate_execute_if_type_exist:nT

To check that a particular object type is valid.

```
\cs_new_protected:Npn \__xtemplate_execute_if_type_exist:nT #1#2
  {
    \prop_if_in:NnTF \g__xtemplate_object_type_prop {#1}
      {#2}
      { \msg_error:nnx { xtemplate } { unknown-object-type } {#1} }
  }
```

__xtemplate_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.

```
\cs_new_protected:Npn \__xtemplate_if_keys_exist:nnT #1#2#3
  {
    \cs_if_exist:cTF { \c__xtemplate_keytypes_root_tl #1 / #2 }
      {#3}
      {
        \msg_error:nnxx { xtemplate } { unknown-template }
          {#1} {#2}
      }
   }
```

__xtemplate_if_key_value:nT
__xtemplate_if_key_value:oT

Tests for the first token in a string being \KeyValue, where \EvaluateNow is not important.

```
\prg_new_conditional:Npnn \__xtemplate_if_key_value:n #1 { T }
  {
    \str_if_eq:noTF { \KeyValue } { \tl_head:w #1 \q_nil \q_stop }
      {\prg_return_true: }
      { \prg_return_false: }
  }
\cs_generate_variant:Nn \__xtemplate_if_key_value:nT { o }
```

__xtemplate_if_eval_now:nTF

Tests for the first token in a string being \EvaluateNow.

```
\prg_new_conditional:Npnn \__xtemplate_if_eval_now:n #1 { TF }
  {
    \str_if_eq:noTF { \EvaluateNow } { \tl_head:w #1 \q_nil \q_stop }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

__xtemplate_if_instance_exist:nnnTF

Testing for an instance is collection dependent.

```
\prg_new_conditional:Npnn \__xtemplate_if_instance_exist:nnn #1#2#3
  { T, F, TF }
  {
    \cs_if_exist:cTF { \c__xtemplate_instances_root_tl #1 / #2 / #3 }
      { \prg_return_true: }
      { \prg_return_false: }
 }
```

__xtemplate_if_use_template:nTF

Tests for the first token in a string being \UseTemplate.

```
\prg_new_conditional:Npnn \__xtemplate_if_use_template:n #1 { TF }
  {
    \str_if_eq:noTF { \UseTemplate } { \tl_head:w #1 \q_nil \q_stop }
      { \prg_return_true: }
      { \prg_return_false: }
}
```

11.4 Saving and recovering property lists

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

__xtemplate_store_defaults:n
__xtemplate_store_keytypes:n
__xtemplate_store_restrictions:n
__xtemplate_store_values:n
__xtemplate_store_vars:n

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

```
\cs_new_protected:Npn \__xtemplate_store_defaults:n #1
  {
    \prop_gclear_new:c { \c__xtemplate_defaults_root_tl #1 }
    \prop_gset_eq:cN { \c__xtemplate_defaults_root_tl #1 }
      \l__xtemplate_values_prop
  }
\cs_new_protected:Npn \__xtemplate_store_keytypes:n #1
  {
    \prop_gclear_new:c { \c__xtemplate_keytypes_root_tl #1 }
    \prop_gset_eq:cN { \c__xtemplate_keytypes_root_tl #1 }
      \l__xtemplate_keytypes_prop
    \seq_gclear_new:c { \c__xtemplate_key_order_root_tl #1 }
    \seq_gset_eq:cN { \c__xtemplate_key_order_root_tl #1 }
      \l__xtemplate_key_order_seq
  }
\cs_new_protected:Npn \__xtemplate_store_values:n #1
  {
    \prop_clear_new:c { \c__xtemplate_values_root_tl #1 }
    \prop_set_eq:cN { \c__xtemplate_values_root_tl #1 }
      \l__xtemplate_values_prop
  }
\cs_new_protected:Npn \__xtemplate_store_restrictions:n #1
  {
    \clist_gclear_new:c { \c__xtemplate_restrict_root_tl #1 }
    \clist_gset_eq:cN { \c__xtemplate_restrict_root_tl #1 }
      \l__xtemplate_restrict_clist
  }
\cs_new_protected:Npn \__xtemplate_store_vars:n #1
  {
    \prop_gclear_new:c { \c__xtemplate_vars_root_tl #1 }
    \prop_gset_eq:cN { \c__xtemplate_vars_root_tl #1 }
      \l__xtemplate_vars_prop
  }
```

__xtemplate_recover_defaults:n
__xtemplate_recover_keytypes:n
__xtemplate_recover_restrictions:n
__xtemplate_recover_values:n
__xtemplate_recover_vars:n

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.

```
\cs_new_protected:Npn \__xtemplate_recover_defaults:n #1
  {
    \prop_set_eq:Nc \l__xtemplate_values_prop
      { \c__xtemplate_defaults_root_tl #1 }
  }
\cs_new_protected:Npn \__xtemplate_recover_keytypes:n #1
  {
    \prop_set_eq:Nc \l__xtemplate_keytypes_prop
      { \c__xtemplate_keytypes_root_tl #1 }
    \seq_set_eq:Nc \l__xtemplate_key_order_seq
      { \c__xtemplate_key_order_root_tl #1 }
  }
\cs_new_protected:Npn \__xtemplate_recover_restrictions:n #1
  {
    \clist_set_eq:Nc \l__xtemplate_restrict_clist
      { \c__xtemplate_restrict_root_tl #1 }
  }
\cs_new_protected:Npn \__xtemplate_recover_values:n #1
  {
    \prop_set_eq:Nc \l__xtemplate_values_prop
      { \c__xtemplate_values_root_tl #1 }
  }
\cs_new_protected:Npn \__xtemplate_recover_vars:n #1
  {
    \prop_set_eq:Nc \l__xtemplate_vars_prop
      { \c__xtemplate_vars_root_tl #1 }
  }
```

11.5 Creating new object types

__xtemplate_declare_object_type:nn

Although the object 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 object type.

```
\cs_new_protected:Npn \__xtemplate_declare_object_type:nn #1#2
  {
    \int_set:Nn \l__xtemplate_tmp_int {#2}
    \bool_if:nTF
      {
        \int_compare_p:nNn {#2} > \c_nine ||
        \int_compare_p:nNn {#2} < \c_zero
      }
      {
        \msg_error:nnxx { xtemplate } { bad-number-of-arguments }
          {#1} { \exp_not:V \l__xtemplate_tmp_int }
      }
      {
        \msg_info:nnxx { xtemplate } { declare-object-type }
          {#1} {#2}
        \prop_gput:NnV \g__xtemplate_object_type_prop {#1}
          \l__xtemplate_tmp_int
      }
  }
```

11.6 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.

__xtemplate_declare_template_keys:nnnn

The main function for the "design" part of creating a template starts by checking that the object 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 of to the storage routine to save the parsed information properly.

```
\cs_new_protected:Npn \__xtemplate_declare_template_keys:nnnn #1#2#3#4
  {
    \__xtemplate_execute_if_type_exist:nT {#1}
      {
        \__xtemplate_execute_if_arg_agree:nnT {#1} {#3}
          {
            \prop_clear:N \l__xtemplate_values_prop
            \prop_clear:N \l__xtemplate_keytypes_prop
            \seq_clear:N \l__xtemplate_key_order_seq
            \keyval_parse:NNn
              \__xtemplate_parse_keys_elt:n \__xtemplate_parse_keys_elt:nn {#4}
            \__xtemplate_store_defaults:n { #1 / #2 }
            \__xtemplate_store_keytypes:n { #1 / #2 }
          }
      }
  }
```

__xtemplate_parse_keys_elt:n
__xtemplate_parse_keys_elt_aux:n
__xtemplate_parse_keys_elt_aux:

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.

```
\cs_new_protected:Npn \__xtemplate_parse_keys_elt:n #1
  {
    \__xtemplate_split_keytype:n {#1}
    \bool_if:NF \l__xtemplate_error_bool
      {
        \__xtemplate_execute_if_keytype_exist:oT \l__xtemplate_keytype_tl
          {
            \seq_map_function:NN \c__xtemplate_keytypes_arg_seq
              \__xtemplate_parse_keys_elt_aux:n
            \bool_if:NF \l__xtemplate_error_bool
              {
                \seq_if_in:NoTF \l__xtemplate_key_order_seq
                  \l__xtemplate_key_name_tl
                  {
                    \msg_error:nnx { xtemplate }
                      { duplicate-key-interface }
                      { \l__xtemplate_key_name_tl }
                  }
                  { \__xtemplate_parse_keys_elt_aux: }
              }
          }
      }
  }
\cs_new_protected_nopar:Npn \__xtemplate_parse_keys_elt_aux:n #1
  {
    \str_if_eq:onT \l__xtemplate_keytype_tl {#1}
      {
        \tl_if_empty:NT \l__xtemplate_keytype_arg_tl
          {
            \msg_error:nnx { xtemplate }
              { keytype-requires-argument } {#1}
            \bool_set_true:N \l__xtemplate_error_bool
            \seq_map_break:
          }
      }
  }
\cs_new_nopar:Npn \__xtemplate_parse_keys_elt_aux:
  {
    \tl_set:Nx \l__xtemplate_tmp_tl
      {
        \l__xtemplate_keytype_tl
        \tl_if_empty:NF \l__xtemplate_keytype_arg_tl
          { { \l__xtemplate_keytype_arg_tl } }
      }
    \prop_put:Noo \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
      \l__xtemplate_tmp_tl
    \seq_put_right:No \l__xtemplate_key_order_seq \l__xtemplate_key_name_tl
    \str_if_eq:onT \l__xtemplate_keytype_tl { choice }
      {
        \clist_if_in:NnT \l__xtemplate_keytype_arg_tl { unknown }
          { \msg_error:nn { xtemplate } { choice-unknown-reserved } }
      }
  }
```

__xtemplate_parse_keys_elt:nn

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

```
\cs_new_protected:Npn \__xtemplate_parse_keys_elt:nn #1#2
  {
    \__xtemplate_parse_keys_elt:n {#1}
    \use:c { __xtemplate_store_value_ \l__xtemplate_keytype_tl :n } {#2}
  }
```

__xtemplate_split_keytype:n
__xtemplate_split_keytype_aux:w

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

```
\group_begin:
\char_set_lccode:nn { `\@ } { `\: }
\char_set_catcode_other:N \@
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new_protected:Npn \__xtemplate_split_keytype:n #1
      {
        \bool_set_false:N \l__xtemplate_error_bool
        \tl_set:Nn \l__xtemplate_tmp_tl {#1}
        \tl_remove_all:Nn \l__xtemplate_tmp_tl { ~ }
        \tl_replace_all:Nnn \l__xtemplate_tmp_tl { : } { @ }
        \tl_if_in:onTF \l__xtemplate_tmp_tl { @ }
          {
            \tl_clear:N \l__xtemplate_key_name_tl
            \exp_after:wN \__xtemplate_split_keytype_aux:w
              \l__xtemplate_tmp_tl \q_stop
          }
          {
            \bool_set_true:N \l__xtemplate_error_bool
            \msg_error:nnx { xtemplate } { missing-keytype } {#1}
          }
      }
    \cs_new_protected:Npn \__xtemplate_split_keytype_aux:w #1 @ #2 \q_stop
      {
        \tl_put_right:Nx \l__xtemplate_key_name_tl { \tl_to_str:n {#1} }
        \tl_if_in:nnTF {#2} { @ }
          {
            \tl_put_right:Nn \l__xtemplate_key_name_tl { @ }
            \__xtemplate_split_keytype_aux:w #2 \q_stop
          }
          {
            \tl_if_empty:NTF \l__xtemplate_key_name_tl
              { \msg_error:nnx { xtemplate } { empty-key-name } { @ #2 } }
              { \__xtemplate_split_keytype_arg:n {#2} }
          }
      }
  }
```

__xtemplate_split_keytype_arg:n
__xtemplate_split_keytype_arg:o
__xtemplate_split_keytype_arg_aux:n
__xtemplate_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).

```
\cs_new_protected:Npn \__xtemplate_split_keytype_arg:n #1
  {
    \tl_set:Nn \l__xtemplate_keytype_tl {#1}
    \tl_clear:N \l__xtemplate_keytype_arg_tl
    \cs_set_protected_nopar:Npn \__xtemplate_split_keytype_arg_aux:n ##1
      {
        \tl_if_in:nnT {#1} {##1}
          {
            \cs_set:Npn \__xtemplate_split_keytype_arg_aux:w
              ####1 ##1 ####2 \q_stop
              {
                \tl_if_empty:nT {####1}
                  {
                    \tl_set:Nn \l__xtemplate_keytype_tl {##1}
                    \tl_set:Nn \l__xtemplate_keytype_arg_tl {####2}
                    \seq_map_break:
                  }
              }
            \__xtemplate_split_keytype_arg_aux:w #1 \q_stop
          }
      }
    \seq_map_function:NN \c__xtemplate_keytypes_arg_seq
      \__xtemplate_split_keytype_arg_aux:n
  }
\cs_generate_variant:Nn \__xtemplate_split_keytype_arg:n { o }
\cs_new_nopar:Npn \__xtemplate_split_keytype_arg_aux:n #1 { }
\cs_new_nopar:Npn \__xtemplate_split_keytype_arg_aux:w #1 \q_stop { }
```

(*End definition for* `\l__xtemplate_default_tl` *This function is documented on page* **??**.)

11.6.1 Storing values

As xtemplate 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.

__xtemplate_store_value_boolean:n

Storing Boolean values requires a test for delayed evaluation, but is different to the various numerical variable types as there are only two possible values to store. So the code here tests the default switch and then records the meaning (either `true` or `false`).

```
\cs_new_protected:Npn \__xtemplate_store_value_boolean:n #1
  {
    \__xtemplate_if_eval_now:nTF {#1}
      {
        \bool_if:cTF { c_ #1 _bool }
          {
            \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl
              { true }
          }
          {
            \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl
              { false }
          }
      }
      {
        \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
      }
  }
```

(*End definition for* `\__xtemplate_store_value_boolean:n` *This function is documented on page* **??**.)

__xtemplate_store_value_code:n
__xtemplate_store_value_choice:n
__xtemplate_store_value_commalist:n
__xtemplate_store_value_function:n
__xtemplate_store_value_instance:n
__xtemplate_store_value_real:n
__xtemplate_store_value_tokenlist:n

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

```
\cs_new_protected:Npn \__xtemplate_store_value_code:n #1
  { \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1} }
\cs_new_eq:NN \__xtemplate_store_value_choice:n    \__xtemplate_store_value_code:n
\cs_new_eq:NN \__xtemplate_store_value_commalist:n \__xtemplate_store_value_code:n
\cs_new_eq:NN \__xtemplate_store_value_function:n  \__xtemplate_store_value_code:n
\cs_new_eq:NN \__xtemplate_store_value_instance:n  \__xtemplate_store_value_code:n
\cs_new_eq:NN \__xtemplate_store_value_real:n      \__xtemplate_store_value_code:n
\cs_new_eq:NN \__xtemplate_store_value_tokenlist:n \__xtemplate_store_value_code:n
```

(*End definition for* `\__xtemplate_store_value_code:n` *This function is documented on page* **??**.)

__xtemplate_store_value_integer:n
__xtemplate_store_value_length:n
__xtemplate_store_value_muskip:n
__xtemplate_store_value_skip:n

Storing the value of a number is in all cases more or less the same. If evaluation is taking place now, assignment is made to a scratch variable, and this result is then stored. On the other hand, if evaluation is delayed the current data is simply stored "as is".

```
\cs_new_protected:Npn \__xtemplate_store_value_integer:n #1
  {
    \__xtemplate_if_eval_now:nTF {#1}
      {
        \int_set:Nn \l__xtemplate_tmp_int {#1}
        \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_int
          \l__xtemplate_tmp_int
      }
      {
        \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
      }
  }
\cs_new_protected:Npn \__xtemplate_store_value_length:n #1
  {
    \__xtemplate_if_eval_now:nTF {#1}
      {
        \dim_set:Nn \l__xtemplate_tmp_dim {#1}
        \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_tl
          \l__xtemplate_tmp_dim
      }
      {
        \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
      }
  }
\cs_new_protected:Npn \__xtemplate_store_value_muskip:n #1
  {
    \__xtemplate_if_eval_now:nTF {#1}
      {
        \muskip_set:Nn \l__xtemplate_tmp_muskip {#1}
        \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_tl
          \l__xtemplate_tmp_muskip
      }
      {
        \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
      }
  }
\cs_new_protected:Npn \__xtemplate_store_value_skip:n #1
  {
    \__xtemplate_if_eval_now:nTF {#1}
      {
        \skip_set:Nn \l__xtemplate_tmp_skip {#1}
        \prop_put:NVV \l__xtemplate_values_prop \l__xtemplate_key_name_tl
          \l__xtemplate_tmp_skip
      }
      {
        \prop_put:Non \l__xtemplate_values_prop \l__xtemplate_key_name_tl {#1}
      }
  }
```

(*End definition for* `\__xtemplate_store_value_integer:n` *This function is documented on page* **??**.)

11.7 Implementation part of template declaration

`\__xtemplate_declare_template_code:nnnnn` 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.

```
\cs_new_protected:Npn \__xtemplate_declare_template_code:nnnnn #1#2#3#4#5
  {
    \__xtemplate_execute_if_type_exist:nT {#1}
      {
        \__xtemplate_execute_if_arg_agree:nnT {#1}{#3}
         {
          \__xtemplate_if_keys_exist:nnT {#1} {#2}
            {
              \__xtemplate_store_key_implementation:nnn {#1} {#2} {#4}
              \cs_generate_from_arg_count:cNnn
                { \c__xtemplate_code_root_tl #1 / #2 }
                \cs_gset_protected:Npn {#3} {#5}
            }
         }
      }
  }
```

(*End definition for* `\__xtemplate_declare_template_code:nnnnn` *This function is documented on page* **??**.)

`\__xtemplate_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.

```
\cs_new_protected:Npn \__xtemplate_store_key_implementation:nnn #1#2#3
  {
    \__xtemplate_recover_defaults:n { #1 / #2 }
    \__xtemplate_recover_keytypes:n { #1 / #2 }
    \prop_clear:N \l__xtemplate_vars_prop
    \keyval_parse:NNn
      \__xtemplate_parse_vars_elt:n \__xtemplate_parse_vars_elt:nn {#3}
    \__xtemplate_store_vars:n { #1 / #2 }
    \clist_clear:N \l__xtemplate_restrict_clist
    \__xtemplate_store_restrictions:n { #1 / #2 }
    \prop_map_inline:Nn \l__xtemplate_keytypes_prop
      {
        \msg_error:nnxxx { xtemplate } { key-not-implemented }
          {##1} {#2} {#1}
      }
  }
```

(*End definition for* `\__xtemplate_store_key_implementation:nnn` *This function is documented on page* **??**.)

__xtemplate_parse_vars_elt:n

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

```
\cs_new_protected:Npn \__xtemplate_parse_vars_elt:n #1
  { \msg_error:nnx { xtemplate } { key-no-variable } {#1} }
```

(*End definition for* `\__xtemplate_parse_vars_elt:n` *This function is documented on page* **??**.)

__xtemplate_parse_vars_elt:nn

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.

```
\cs_new_protected:Npn \__xtemplate_parse_vars_elt:nn #1#2
 {
    \tl_set:Nx \l__xtemplate_key_name_tl { \tl_to_str:n {#1} }
    \tl_remove_all:Nn \l__xtemplate_key_name_tl { ~ }
    \prop_get:NoNTF
      \l__xtemplate_keytypes_prop
      \l__xtemplate_key_name_tl
      \l__xtemplate_keytype_tl
      {
        \__xtemplate_split_keytype_arg:o \l__xtemplate_keytype_tl
        \__xtemplate_parse_vars_elt_aux:n {#2}
        \prop_remove:NV \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
      }
      { \msg_error:nnx { xtemplate } { unknown-key } {#1} }
  }
```

(*End definition for* `\__xtemplate_parse_vars_elt:nn` *This function is documented on page* **??**.)

__xtemplate_parse_vars_elt_aux:n
__xtemplate_parse_vars_elt_aux:w

There now needs to be some sanity checking on the variable name given. This does not apply for `choice` or `code` "variables", but in all other cases the variable needs to exist. Also, the only prefix acceptable is `global`. So there are a few related checks to make.

```
\cs_new_protected:Npn \__xtemplate_parse_vars_elt_aux:n #1
  {
    \str_if_eq:onTF \l__xtemplate_keytype_tl { choice }
      { \__xtemplate_implement_choices:n {#1} }
      {
        \str_if_eq:onTF \l__xtemplate_keytype_tl { code }
          {
            \prop_put:Non \l__xtemplate_vars_prop
              \l__xtemplate_key_name_tl {#1}
          }
          {
            \tl_if_single:nTF {#1}
              {
                \cs_if_exist:NF #1
                  { \__xtemplate_create_variable:N #1 }
                \prop_put:Non \l__xtemplate_vars_prop
                  \l__xtemplate_key_name_tl {#1}
            }
            {
              \tl_if_in:nnTF {#1} { global }
                { \__xtemplate_parse_vars_elt_aux:w #1 \q_stop }
                {
                  \msg_error:nnx { xtemplate } { bad-variable }
                    { \tl_to_str:n {#1} }
                }
            }
        }
    }
  }
\cs_new_protected:Npn \__xtemplate_parse_vars_elt_aux:w #1 global #2 \q_stop
  {
    \tl_if_empty:nTF {#1}
      {
        \tl_if_single:nTF {#2}
          {
            \cs_if_exist:NF #2
              { \__xtemplate_create_variable:N #2 }
            \prop_put:Non \l__xtemplate_vars_prop
              \l__xtemplate_key_name_tl { #1 global #2 }
          }
          {
            \msg_error:nnx { xtemplate } { bad-variable }
              { \tl_to_str:n { #1 global #2 } }
          }
      }
      {
          \msg_error:nnx { xtemplate } { bad-variable }
            { \tl_to_str:n { #1 global #2 } }
      }
  }
```

(*End definition for* `\__xtemplate_parse_vars_elt_aux:n` *This function is documented on page* **??**.)

`\__xtemplate_create_variable:N` A shortcut to create non-declared variables. Some types need a name mapping, others can be used directly.

```
\cs_new_protected_nopar:Npn \__xtemplate_create_variable:N #1
  {
    \str_case:onn \l__xtemplate_keytype_tl
      {
        { boolean }   { \bool_new:N #1 }
        { commalist } { \clist_new:N #1 }
        { function }  { \cs_new:Npn #1 { } }
        { instance }  { \cs_new_protected:Npn #1 { } }
        { integer }   { \int_new:N #1 }
        { length }    { \dim_new:N #1 }
        { real }      { \fp_new:N #1 }
        { tokenlist } { \tl_new:N #1 }
      }
      { \use:c { \l__xtemplate_keytype_tl _ new:N } #1 }
  }
```

(*End definition for* `\__xtemplate_create_variable:N` *This function is documented on page* **??**.)

`\__xtemplate_implement_choices:n`
`\__xtemplate_implement_choices_default:`

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

```
\cs_new_protected:Npn \__xtemplate_implement_choices:n #1
  {
    \clist_set_eq:NN \l__xtemplate_tmp_clist \l__xtemplate_keytype_arg_tl
    \prop_put:Non \l__xtemplate_vars_prop \l__xtemplate_key_name_tl { }
    \keyval_parse:NNn
      \__xtemplate_implement_choice_elt:n \__xtemplate_implement_choice_elt:nn
      {#1}
    \prop_get:NoNT \l__xtemplate_values_prop \l__xtemplate_key_name_tl
      \l__xtemplate_tmp_tl
      { \__xtemplate_implement_choices_default: }
    \clist_if_empty:NF \l__xtemplate_tmp_clist
      {
        \clist_map_inline:Nn \l__xtemplate_tmp_clist
          {
            \msg_error:nnx { xtemplate } { choice-not-implemented }
              {##1}
          }
      }
  }
```

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

```
\cs_new_protected_nopar:Npn \__xtemplate_implement_choices_default:
  {
    \tl_set:Nx \l__xtemplate_tmp_tl
      { \l__xtemplate_key_name_tl \c_space_tl \l__xtemplate_tmp_tl }
    \prop_if_in:NoF \l__xtemplate_vars_prop \l__xtemplate_tmp_tl
      {
        \tl_set:Nx \l__xtemplate_tmp_tl
          { \l__xtemplate_key_name_tl \c_space_tl \l__xtemplate_tmp_tl }
        \prop_if_in:NoF \l__xtemplate_vars_prop \l__xtemplate_tmp_tl
          {
            \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
              \l__xtemplate_tmp_tl
            \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
            \prop_get:NoN \l__xtemplate_values_prop \l__xtemplate_key_name_tl
              \l__xtemplate_tmp_tl
            \msg_error:nnxxx { xtemplate } { unknown-default-choice }
              { \l__xtemplate_key_name_tl } { \l__xtemplate_key_name_tl }
              { \l__xtemplate_keytype_arg_tl }
          }
      }
  }
```

(*End definition for* `\__xtemplate_implement_choices:n` *This function is documented on page* **??**.)

`\__xtemplate_implement_choice_elt:n`
`\__xtemplate_implement_choice_elt:nn`

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.

```
\cs_new_protected:Npn \__xtemplate_implement_choice_elt:n #1
  {
    \clist_if_empty:NTF \l__xtemplate_tmp_clist
      {
        \str_if_eq:nnF {#1} { unknown }
          {
            \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
              \l__xtemplate_tmp_tl
            \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
            \msg_error:nnxxx { xtemplate } { unknown-choice }
              { \l__xtemplate_key_name_tl } {#1}
              { \l__xtemplate_keytype_arg_tl }
          }
      }
      {
        \clist_if_in:NnTF \l__xtemplate_tmp_clist {#1}
          { \clist_remove_all:Nn \l__xtemplate_tmp_clist {#1} }
          {
            \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
              \l__xtemplate_tmp_tl
            \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
            \msg_error:nnxxx { xtemplate } { unknown-choice }
              { \l__xtemplate_key_name_tl } {#1}
              { \l__xtemplate_keytype_arg_tl }
          }
      }
  }
\cs_new_protected:Npn \__xtemplate_implement_choice_elt:nn #1#2
  {
    \__xtemplate_implement_choice_elt:n {#1}
    \tl_set:Nx \l__xtemplate_tmp_tl
      { \l__xtemplate_key_name_tl \c_space_tl #1 }
    \prop_put:Non \l__xtemplate_vars_prop \l__xtemplate_tmp_tl {#2}
  }
```

(*End definition for* `\__xtemplate_implement_choice_elt:n` *This function is documented on page* **??**.)

11.8 Editing template defaults

Template defaults can be edited either with no other changes or to prevent further editing, forming a "restricted template". In the later case, a new template results, whereas simple editing does not produce a new template name.

__xtemplate_declare_restricted:nnnn

Creating a restricted template means copying the old template to the new one first.

```
\cs_new_protected:Npn \__xtemplate_declare_restricted:nnnn #1#2#3#4
  {
    \__xtemplate_if_keys_exist:nnT {#1} {#2}
      {
        \__xtemplate_set_template_eq:nn { #1 / #3 } { #1 / #2 }
        \bool_set_true:N \l__xtemplate_restrict_bool
        \__xtemplate_edit_defaults_aux:nnn {#1} {#3} {#4}
      }
  }
```

(*End definition for* `\__xtemplate_declare_restricted:nnnn` *This function is documented on page* **??**.)

__xtemplate_edit_defaults:nnn
__xtemplate_edit_defaults_aux: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. The auxiliary function is used to allow code-sharing with the template-restriction system.

```
\cs_new_protected:Npn \__xtemplate_edit_defaults:nnn
  {
    \bool_set_false:N \l__xtemplate_restrict_bool
    \__xtemplate_edit_defaults_aux:nnn
  }
\cs_new_protected:Npn \__xtemplate_edit_defaults_aux:nnn #1#2#3
  {
    \__xtemplate_if_keys_exist:nnT {#1} {#2}
      {
        \__xtemplate_recover_defaults:n { #1 / #2 }
        \__xtemplate_recover_restrictions:n { #1 / #2 }
        \__xtemplate_parse_values:nn { #1 / #2 } {#3}
        \__xtemplate_store_defaults:n { #1 / #2 }
        \__xtemplate_store_restrictions:n { #1 / #2 }
      }
  }
```

(*End definition for* `\__xtemplate_edit_defaults:nnn` *This function is documented on page* **??**.)

__xtemplate_parse_values:nn

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.

```
\cs_new_protected:Npn \__xtemplate_parse_values:nn #1#2
  {
    \__xtemplate_recover_keytypes:n {#1}
    \clist_clear:N \l__xtemplate_restrict_clist
    \keyval_parse:NNn
      \__xtemplate_parse_values_elt:n \__xtemplate_parse_values_elt:nn {#2}
  }
```

(*End definition for* `\__xtemplate_parse_values:nn` *This function is documented on page* **??**.)

__xtemplate_parse_values_elt:n

Every key needs a value, so this is just an error routine.

```
\cs_new_protected:Npn \__xtemplate_parse_values_elt:n #1
  {
    \bool_set_true:N \l__xtemplate_error_bool
    \msg_error:nnx { xtemplate } { key-no-value } {#1}
  }
```
(*End definition for* `\__xtemplate_parse_values_elt:n` *This function is documented on page* **??**.)

`\__xtemplate_parse_values_elt:nn`
`\__xtemplate_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__xtemplate_key_name_tl`. When a template is being restricted, the setting code will be skipped for restricted keys.

```
\cs_new_protected:Npn \__xtemplate_parse_values_elt:nn #1#2
  {
    \tl_set:Nx \l__xtemplate_key_name_tl { \tl_to_str:n {#1} }
    \tl_remove_all:Nn \l__xtemplate_key_name_tl { ~ }
    \prop_get:NoNTF \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
      \l__xtemplate_tmp_tl
      {
        \bool_if:NTF \l__xtemplate_restrict_bool
          {
            \clist_if_in:NoF \l__xtemplate_restrict_clist
              \l__xtemplate_key_name_tl
                { \__xtemplate_parse_values_elt_aux:n {#2} }
          }
          { \__xtemplate_parse_values_elt_aux:n {#2} }
      }
      {
        \msg_error:nnx { xtemplate } { unknown-key }
          { \l__xtemplate_key_name_tl }
      }
  }
\cs_new_protected:Npn \__xtemplate_parse_values_elt_aux:n #1
  {
    \clist_put_right:No \l__xtemplate_restrict_clist \l__xtemplate_key_name_tl
    \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
    \use:c { __xtemplate_store_value_ \l__xtemplate_keytype_tl :n } {#1}
  }
```
(*End definition for* `\__xtemplate_parse_values_elt:nn` *This function is documented on page* **??**.)

`\__xtemplate_set_template_eq:nn`

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.

```
\cs_new_protected:Npn \__xtemplate_set_template_eq:nn #1#2
  {
    \__xtemplate_recover_defaults:n {#2}
    \__xtemplate_store_defaults:n {#1}
    \__xtemplate_recover_keytypes:n {#2}
    \__xtemplate_store_keytypes:n {#1}
    \__xtemplate_recover_vars:n {#2}
    \__xtemplate_store_vars:n {#1}
    \cs_gset_eq:cc { \c__xtemplate_code_root_tl #1 }
      { \c__xtemplate_code_root_tl #2 }
  }
```
(*End definition for* `\__xtemplate_set_template_eq:nn` *This function is documented on page* **??**.)

11.9 Creating instances of templates

__xtemplate_declare_instance:nnnnn
__xtemplate_declare_instance_aux:nnnnn

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. A final check is also made so that there is always an instance "outside" of any collection.

```
\cs_new_protected:Npn \__xtemplate_declare_instance:nnnnn #1#2#3#4#5
  {
    \__xtemplate_execute_if_code_exist:nnT {#1} {#2}
      {
        \__xtemplate_recover_defaults:n { #1 / #2 }
        \__xtemplate_recover_vars:n { #1 / #2 }
        \__xtemplate_declare_instance_aux:nnnnn {#1} {#2} {#3} {#4} {#5}
      }
  }
\cs_new_protected:Npn \__xtemplate_declare_instance_aux:nnnnn #1#2#3#4#5
  {
    \bool_set_false:N \l__xtemplate_error_bool
    \__xtemplate_parse_values:nn { #1 / #2 } {#5}
    \bool_if:NF \l__xtemplate_error_bool
      {
        \prop_put:Nnn \l__xtemplate_values_prop { from~template } {#2}
        \__xtemplate_store_values:n { #1 / #3 / #4 }
        \__xtemplate_convert_to_assignments:
        \cs_set_protected:cpx { \c__xtemplate_instances_root_tl #1 / #3 / #4 }
          {
            \exp_not:N \__xtemplate_assignments_push:n
              { \exp_not:o \l__xtemplate_assignments_tl }
            \exp_not:c { \c__xtemplate_code_root_tl #1 / #2 }
          }
        \__xtemplate_if_instance_exist:nnnF {#1} { } {#4}
          {
            \cs_set_eq:cc
              { \c__xtemplate_instances_root_tl #1 /    / #4 }
              { \c__xtemplate_instances_root_tl #1 / #3 / #4 }
          }
      }
  }
```

(*End definition for* `\__xtemplate_declare_instance:nnnnn` *This function is documented on page* **??**.)

__xtemplate_edit_instance:nnnn
__xtemplate_edit_instance_aux:nnnnn
__xtemplate_edit_instance_aux:nonnn

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.

```
\cs_new_protected:Npn \__xtemplate_edit_instance:nnnn #1#2#3
  {
    \__xtemplate_if_instance_exist:nnnTF {#1} {#2} {#3}
      {
        \__xtemplate_recover_values:n { #1 / #2 / #3 }
        \prop_get:NnN \l__xtemplate_values_prop { from~template }
          \l__xtemplate_tmp_tl
        \__xtemplate_edit_instance_aux:nonnn {#1} \l__xtemplate_tmp_tl
          {#2} {#3}
      }
      {
        \msg_error:nnxx { xtemplate } { unknown-instance }
          {#1} {#3}
      }
  }
\cs_new_protected:Npn \__xtemplate_edit_instance_aux:nnnnn #1#2
  {
    \__xtemplate_recover_vars:n { #1 / #2 }
    \__xtemplate_declare_instance_aux:nnnnn {#1} {#2}
  }
\cs_generate_variant:Nn \__xtemplate_edit_instance_aux:nnnnn { no }
```

(*End definition for* `\__xtemplate_edit_instance:nnnn` *This function is documented on page* **??**.)

__xtemplate_convert_to_assignments:
__xtemplate_convert_to_assignments_aux:n
__xtemplate_convert_to_assignments_aux:nn
__xtemplate_convert_to_assignments_aux:no

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”.

```
\cs_new_protected_nopar:Npn \__xtemplate_convert_to_assignments:
  {
    \tl_clear:N \l__xtemplate_assignments_tl
    \seq_map_function:NN \l__xtemplate_key_order_seq
      \__xtemplate_convert_to_assignments_aux:n
  }
\cs_new_protected:Npn \__xtemplate_convert_to_assignments_aux:n #1
  {
    \prop_get:NnN \l__xtemplate_keytypes_prop {#1} \l__xtemplate_tmp_tl
    \__xtemplate_convert_to_assignments_aux:no {#1} \l__xtemplate_tmp_tl
  }
```

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.

```
\cs_new_protected:Npn \__xtemplate_convert_to_assignments_aux:nn #1#2
  {
    \prop_get:NnNT \l__xtemplate_values_prop {#1} \l__xtemplate_value_tl
      {
        \prop_get:NnNTF \l__xtemplate_vars_prop {#1} \l__xtemplate_var_tl
          {
            \__xtemplate_split_keytype_arg:n {#2}
            \str_if_eq:onF \l__xtemplate_keytype_tl { choice }
              {
                \str_if_eq:onF \l__xtemplate_keytype_tl { code }
                  { \__xtemplate_find_global: }
            }
          \tl_set:Nn \l__xtemplate_key_name_tl {#1}
          \use:c { __xtemplate_assign_ \l__xtemplate_keytype_tl : }
        }
        { \msg_error:nnx { xtemplate } { unknown-attribute } {#1} }
    }
  }
\cs_generate_variant:Nn \__xtemplate_convert_to_assignments_aux:nn { no }
```

(*End definition for* `\__xtemplate_convert_to_assignments:` *This function is documented on page* **??**.)

`\__xtemplate_find_global:`
`\__xtemplate_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.

```
\cs_new_protected_nopar:Npn \__xtemplate_find_global:
  {
    \bool_set_false:N \l__xtemplate_global_bool
    \tl_if_in:onT \l__xtemplate_var_tl { global }
      {
        \exp_after:wN \__xtemplate_find_global_aux:w \l__xtemplate_var_tl \q_stop
      }
  }
\cs_new_protected:Npn \__xtemplate_find_global_aux:w  #1 global #2 \q_stop
  {
    \tl_set:Nn \l__xtemplate_var_tl {#2}
    \bool_set_true:N \l__xtemplate_global_bool
  }
```

(*End definition for* `\__xtemplate_find_global:` *This function is documented on page* **??**.)

11.10 Using templates directly

`\__xtemplate_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.

```
\cs_new_protected:Npn \__xtemplate_use_template:nnn #1#2#3
  {
    \__xtemplate_execute_if_code_exist:nnT {#1} {#2}
      {
        \__xtemplate_recover_defaults:n { #1 / #2 }
        \__xtemplate_recover_vars:n { #1 / #2 }
        \__xtemplate_parse_values:nn { #1 / #2 } {#3}
        \__xtemplate_convert_to_assignments:
        \use:c { \c__xtemplate_code_root_tl #1 / #2  }
      }
  }
```

(*End definition for* `\__xtemplate_use_template:nnn` *This function is documented on page* **??**.)

11.11 Assigning values to variables

__xtemplate_assign_boolean:
__xtemplate_assign_boolean_aux:n

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.

```
\cs_new_protected_nopar:Npn \__xtemplate_assign_boolean:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_boolean_aux:n { bool_gset } }
      { \__xtemplate_assign_boolean_aux:n { bool_set } }
  }
\cs_new_protected_nopar:Npn \__xtemplate_assign_boolean_aux:n #1
  {
    \__xtemplate_if_key_value:oT \l__xtemplate_value_tl
      { \__xtemplate_key_to_value: }
    \tl_put_left:Nx \l__xtemplate_assignments_tl
      {
        \exp_not:c { #1 _ \l__xtemplate_value_tl :N }
        \exp_not:o \l__xtemplate_var_tl
      }
  }
```

(*End definition for* `\__xtemplate_assign_boolean:` *This function is documented on page* **??**.)

__xtemplate_assign_choice:
__xtemplate_assign_choice_aux:n
__xtemplate_assign_choice_aux:o

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

```
\cs_new_protected_nopar:Npn \__xtemplate_assign_choice:
  {
    \__xtemplate_assign_choice_aux:xF
      { \l__xtemplate_key_name_tl \c_space_tl \l__xtemplate_value_tl }
      {
        \__xtemplate_assign_choice_aux:xF
          { \l__xtemplate_key_name_tl \c_space_tl unknown }
          {
            \prop_get:NoN \l__xtemplate_keytypes_prop \l__xtemplate_key_name_tl
              \l__xtemplate_tmp_tl
            \__xtemplate_split_keytype_arg:o \l__xtemplate_tmp_tl
            \msg_error:nnxxx { xtemplate } { unknown-choice }
              { \l__xtemplate_key_name_tl } { \l__xtemplate_value_tl }
              { \l__xtemplate_keytype_arg_tl }
          }
      }
  }
\cs_new_protected_nopar:Npn \__xtemplate_assign_choice_aux:nF #1
  {
    \prop_get:NnNTF
      \l__xtemplate_vars_prop
      {#1}
      \l__xtemplate_tmp_tl
      { \tl_put_right:No \l__xtemplate_assignments_tl \l__xtemplate_tmp_tl }
  }
\cs_generate_variant:Nn \__xtemplate_assign_choice_aux:nF { x }
```

(*End definition for* `\__xtemplate_assign_choice:` *This function is documented on page* **??**.)

`\__xtemplate_assign_code:`
`\__xtemplate_assign_code:n`

Assigning general code to a key needs a scratch function to be created and run when `\AssignTemplateKeys` is called. So the appropriate definition then use is created in the token list variable.

```
\cs_new_protected_nopar:Npn \__xtemplate_assign_code:
  {
    \tl_put_left:Nx \l__xtemplate_assignments_tl
      {
        \cs_set_protected:Npn \__xtemplate_assign_code:n \exp_not:n {##1}
          { \exp_not:o \l__xtemplate_var_tl }
        \__xtemplate_assign_code:n { \exp_not:o \l__xtemplate_value_tl }
      }
  }
\cs_new_protected:Npn \__xtemplate_assign_code:n #1 { }
```

(*End definition for* `\__xtemplate_assign_code:` *This function is documented on page* **??**.)

`\__xtemplate_assign_function:`
`\__xtemplate_assign_function_aux:N`

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

```
\cs_new_protected_nopar:Npn \__xtemplate_assign_function:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_function_aux:N \cs_gset:Npn }
      { \__xtemplate_assign_function_aux:N \cs_set:Npn  }
  }
\cs_new_protected_nopar:Npn \__xtemplate_assign_function_aux:N #1
  {
    \tl_put_left:Nx \l__xtemplate_assignments_tl
      {
        \cs_generate_from_arg_count:NNnn
          \exp_not:o \l__xtemplate_var_tl
          \exp_not:N #1
          { \exp_not:o \l__xtemplate_keytype_arg_tl }
          { \exp_not:o \l__xtemplate_value_tl }
      }
  }
```

(*End definition for* `\__xtemplate_assign_function:` *This function is documented on page* **??**.)

`\__xtemplate_assign_instance:`
`\__xtemplate_assign_instance_aux:N`

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

```
\cs_new_protected_nopar:Npn \__xtemplate_assign_instance:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_instance_aux:N \cs_gset_protected:Npn }
      { \__xtemplate_assign_instance_aux:N \cs_set_protected:Npn  }
  }
\cs_new_protected_nopar:Npn \__xtemplate_assign_instance_aux:N #1
  {
    \tl_put_left:Nx \l__xtemplate_assignments_tl
      {
        \exp_not:N #1 \exp_not:o \l__xtemplate_var_tl
          {
            \__xtemplate_use_instance:nn
              { \exp_not:o \l__xtemplate_keytype_arg_tl }
              { \exp_not:o \l__xtemplate_value_tl }
          }
      }
  }
```

(*End definition for* `\__xtemplate_assign_instance:` *This function is documented on page* **??**.)

`\__xtemplate_assign_integer:`
`\__xtemplate_assign_length:`
`\__xtemplate_assign_muskip:`
`\__xtemplate_assign_real:`
`\__xtemplate_assign_skip:`

All of the calculated assignments use the same underlying code, with only the low-level assignment function changing.

```
\cs_new_protected_nopar:Npn \__xtemplate_assign_integer:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_variable:N \int_gset:Nn }
      { \__xtemplate_assign_variable:N \int_set:Nn  }
  }
\cs_new_protected_nopar:Npn \__xtemplate_assign_length:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_variable:N \dim_gset:Nn }
      { \__xtemplate_assign_variable:N \dim_set:Nn  }
}
\cs_new_protected_nopar:Npn \__xtemplate_assign_muskip:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_variable:N \muskip_gset:Nn }
      { \__xtemplate_assign_variable:N \muskip_set:Nn  }
  }
\cs_new_protected_nopar:Npn \__xtemplate_assign_real:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_variable:N \fp_gset:Nn }
      { \__xtemplate_assign_variable:N \fp_set:Nn  }
  }
\cs_new_protected_nopar:Npn \__xtemplate_assign_skip:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_variable:N \skip_gset:Nn }
      { \__xtemplate_assign_variable:N \skip_set:Nn  }
  }
```

(*End definition for* `\__xtemplate_assign_integer:` *This function is documented on page* **??**.)

`\__xtemplate_assign_tokenlist:`
`\__xtemplate_assign_tokenlist_aux:N`

Storing lists of tokens is easy: no complex calculations and no need to worry about numbers of arguments.

```
\cs_new_protected_nopar:Npn \__xtemplate_assign_tokenlist:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_tokenlist_aux:N \tl_gset:Nn }
      { \__xtemplate_assign_tokenlist_aux:N \tl_set:Nn  }
  }
\cs_new_protected_nopar:Npn \__xtemplate_assign_tokenlist_aux:N #1
  {
    \__xtemplate_if_key_value:oT \l__xtemplate_value_tl
      { \__xtemplate_key_to_value: }
    \tl_put_left:Nx \l__xtemplate_assignments_tl
      {
        #1 \exp_not:o \l__xtemplate_var_tl
          { \exp_not:o \l__xtemplate_value_tl }
      }
  }
```

(*End definition for* `\__xtemplate_assign_tokenlist:` *This function is documented on page* **??**.)

`\__xtemplate_assign_commalist:`

Very similar for commas lists, so some code is shared.

```
\cs_new_protected_nopar:Npn \__xtemplate_assign_commalist:
  {
    \bool_if:NTF \l__xtemplate_global_bool
      { \__xtemplate_assign_tokenlist_aux:N \clist_gset:Nn }
      { \__xtemplate_assign_tokenlist_aux:N \clist_set:Nn  }
  }
```

(*End definition for* `\__xtemplate_assign_commalist:` *This function is documented on page* **??**.)

`\__xtemplate_assign_variable:N`

A general-purpose function for all of the numerical assignments. As long as the value is not coming from another variable, the stored value is simply transferred for output.

```
\cs_new_protected_nopar:Npn \__xtemplate_assign_variable:N #1
  {
    \__xtemplate_if_key_value:oT \l__xtemplate_value_tl
      { \__xtemplate_key_to_value: }
    \tl_put_left:Nx \l__xtemplate_assignments_tl
      {
        #1 \exp_not:o \l__xtemplate_var_tl
         { \exp_not:o \l__xtemplate_value_tl }
      }
  }
```

(*End definition for* `\__xtemplate_assign_variable:N` *This function is documented on page* **??**.)

`\__xtemplate_key_to_value:`
`\__xtemplate_key_to_value_aux:w`

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.

```
\cs_new_protected_nopar:Npn \__xtemplate_key_to_value:
  { \exp_after:wN \__xtemplate_key_to_value_aux:w \l__xtemplate_value_tl }
\cs_new_protected:Npn \__xtemplate_key_to_value_aux:w \KeyValue #1
  {
    \tl_set:Nx \l__xtemplate_tmp_tl { \tl_to_str:n {#1} }
    \tl_remove_all:Nn \l__xtemplate_key_name_tl { ~ }
    \prop_get:NoNF
      \l__xtemplate_vars_prop
      \l__xtemplate_tmp_tl
      \l__xtemplate_value_tl
      {
        \msg_error:nnx { xtemplate } { unknown-attribute }
          { \l__xtemplate_tmp_tl }
      }
  }
```
(*End definition for* `\__xtemplate_key_to_value`*: This function is documented on page* **??**.)

11.12 Using instances

`\__xtemplate_use_instance:nn`
`\__xtemplate_use_instance_aux:nNnnn`
`\__xtemplate_use_instance_aux:nn`

Using an instance is just a question of finding the appropriate function. There is the possibility that a collection instance exists, so this is checked before trying the general instance. If nothing is found, an error is raised. One additional 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.

```
\cs_new_protected:Npn \__xtemplate_use_instance:nn #1#2
  {
    \__xtemplate_if_use_template:nTF {#2}
      { \__xtemplate_use_instance_aux:nNnnn {#1} #2 }
      { \__xtemplate_use_instance_aux:nn {#1} {#2} }
  }
\cs_new_protected:Npn \__xtemplate_use_instance_aux:nNnnn #1#2#3#4#5
  {
    \str_if_eq:nnTF {#1} {#3}
      { \__xtemplate_use_template:nnn {#3} {#4} {#5} }
      { \msg_error:nnxx { xtemplate } { type-mismatch } {#1} {#3} }
}
\cs_new_protected:Npn \__xtemplate_use_instance_aux:nn #1#2
  {
    \__xtemplate_get_collection:n {#1}
    \__xtemplate_if_instance_exist:nnnTF
      {#1} { \l__xtemplate_collection_tl } {#2}
        {
          \use:c
            {
              \c__xtemplate_instances_root_tl #1 /
                \l__xtemplate_collection_tl / #2
            }
        }
        {
          \__xtemplate_if_instance_exist:nnnTF {#1} { } {#2}
            { \use:c { \c__xtemplate_instances_root_tl #1 / / #2 } }
            {
              \msg_error:nnxx { xtemplate } { unknown-instance }
                {#1} {#2}
            }
        }
    }
```
(*End definition for* `\__xtemplate_use_instance:nn` *This function is documented on page* **??**.)

`\__xtemplate_use_collection:nn` Switching to an instance collection is just a question of setting the appropriate list.

```
\cs_new_protected:Npn \__xtemplate_use_collection:nn #1#2
  { \prop_put:Nnn \l__xtemplate_collections_prop {#1} {#2} }
```
(*End definition for* `\__xtemplate_use_collection:nn` *This function is documented on page* **??**.)

`\__xtemplate_get_collection:n` Recovering the collection for a given type is pretty easy: just a read from the list.

```
\cs_new_protected:Npn \__xtemplate_get_collection:n #1
  {
    \prop_get:NnNF \l__xtemplate_collections_prop {#1}
      \l__xtemplate_collection_tl
      { \tl_clear:N \l__xtemplate_collection_tl }
  }
```
(*End definition for* `\__xtemplate_get_collection:n` *This function is documented on page* **??**.)

11.13 Assignment manipulation

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

`\__xtemplate_assignments_pop:` To actually use the assignments.

```
\cs_new_nopar:Npn \__xtemplate_assignments_pop: { \l__xtemplate_assignments_tl }
```
(*End definition for* `\__xtemplate_assignments_pop:` *This function is documented on page* **??**.)

`\__xtemplate_assignments_push:n` Here, the assignments are stored for later use.

```
\cs_new_protected:Npn \__xtemplate_assignments_push:n #1
  { \tl_set:Nn \l__xtemplate_assignments_tl {#1} }
```
(*End definition for* `\__xtemplate_assignments_push:n` *This function is documented on page* **??**.)

11.14 Showing templates and instances

`\__xtemplate_show_code:nn` Showing the code for a template is just a translation of `\cs_show:c`.

```
\cs_new_protected_nopar:Npn \__xtemplate_show_code:nn #1#2
  { \cs_show:c { \c__xtemplate_code_root_tl #1 / #2 } }
```
(*End definition for* `\__xtemplate_show_code:nn` *This function is documented on page* **??**.)

`\__xtemplate_show_defaults:nn`
`\__xtemplate_show_keytypes:nn`
`\__xtemplate_show_vars:nn`
`\__xtemplate_show:Nnnn`

A modified version of the property-list printing code, such that the output refers to templates and instances rather than to the underlying structures.

```
\cs_new_protected_nopar:Npn \__xtemplate_show_defaults:nn #1#2
  {
    \__xtemplate_if_keys_exist:nnT {#1} {#2}
      {
        \__xtemplate_recover_defaults:n { #1 / #2 }
        \__xtemplate_show:Nnnn \l__xtemplate_values_prop
          {#1} {#2} { default~values }
      }
  }
\cs_new_protected_nopar:Npn \__xtemplate_show_keytypes:nn #1#2
  {
    \__xtemplate_if_keys_exist:nnT {#1} {#2}
      {
        \__xtemplate_recover_keytypes:n { #1 / #2 }
        \__xtemplate_show:Nnnn \l__xtemplate_keytypes_prop
          {#1} {#2} { interface }
      }
  }
\cs_new_protected_nopar:Npn \__xtemplate_show_vars:nn #1#2
  {
     \__xtemplate_execute_if_code_exist:nnT {#1} {#2}
      {
        \__xtemplate_recover_vars:n { #1 / #2 }
        \__xtemplate_show:Nnnn \l__xtemplate_vars_prop
          {#1} {#2} { variable~mapping }
      }
  }
\cs_new_protected_nopar:Npn \__xtemplate_show:Nnnn #1#2#3#4
  {
    \__msg_term:nnnnn { xtemplate }
      { \prop_if_empty:NTF #1 { show-no-attribute } { show-attribute } }
      {#2} {#3} {#4}
    \__msg_show_variable:x
      { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
  }
```

(*End definition for* `\__xtemplate_show_defaults:nn`*,* `\__xtemplate_show_keytypes:nn`*, and* `\__xtemplate_show_vars:nn` *These functions are documented on page* **??**.)

`\template_show_values:nnn` Instance values are a little more complex, as there are the collection and template to consider.

```
\cs_new_protected_nopar:Npn \__xtemplate_show_values:nnn #1#2#3
  {
    \__xtemplate_if_instance_exist:nnnT {#1} {#2} {#3}
      {
        \__xtemplate_recover_values:n { #1 / #2 / #3 }
        \prop_if_empty:NTF \l__xtemplate_values_prop
          {
            \__msg_term:nnnnn { xtemplate } { show-no-values }
              {#1} {#2} {#3}
            \__msg_show_variable:x { }
          }
          {
            \prop_pop:NnN \l__xtemplate_values_prop { from~template }
              \l__xtemplate_tmp_tl
            \__msg_term:nnnnnV { xtemplate } { show-values }
              {#1} {#2} {#3} \l__xtemplate_tmp_tl
            \__msg_show_variable:x
```

```
              {
                \prop_map_function:NN \l__xtemplate_values_prop
                  \__msg_show_item_unbraced:nn
              }
          }
      }
  }
```

(*End definition for* `\template_show_values:nnn` *This function is documented on page* **??**.)

11.15 Messages

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

```
\msg_new:nnnn { xtemplate } { argument-number-mismatch }
  { Object~type~'#1'~takes~#2~argument(s). }
  {
    \c_msg_coding_error_text_tl
    Objects~of~type~'#1'~require~#2~argument(s).\\
    You~have~tried~to~make~a~template~for~'#1'~
    with~#3~argument(s),~which~is~not~possible:~
    the~number~of~arguments~must~agree.
  }
\msg_new:nnnn { xtemplate } { bad-number-of-arguments }
  { Bad~number~of~arguments~for~object~type~'#1'. }
  {
    \c_msg_coding_error_text_tl
    An~object~may~accept~between~0~and~9~arguments.\\
    You~asked~to~use~#2~arguments:~this~is~not~supported.
  }
\msg_new:nnnn { xtemplate } { bad-variable }
  { Incorrect~variable~description~'#1'. }
  {
    The~argument~'#1'~is~not~of~the~form \\
    ~~'<variable>'\\
    ~or~\\
    ~~'global~<variable>'.\\
    It~must~be~given~in~one~of~these~formats~to~be~used~in~a~template.
  }
\msg_new:nnnn { xtemplate } { choice-not-implemented }
  { The~choice~'#1'~has~no~implementation. }
  {
    Each~choice~listed~in~the~interface~for~a~template~must~
    have~an~implementation.
  }
\msg_new:nnnn { xtemplate } { choice-no-code }
  { The~choice~'#1'~requires~implementation~details. }
  {
    \c_msg_coding_error_text_tl
    When~creating~template~code~using~\DeclareTemplateCode,~
    each~choice~name~must~have~an~associated~implementation.\\
    This~should~be~given~after~a~'='~sign:~LaTeX~did~not~find~one.
  }
\msg_new:nnnn { xtemplate } { duplicate-key-interface }
  { Key~'#1'~appears~twice~in~interface~definition~\msg_line_context:. }
  {
    \c_msg_coding_error_text_tl
    Each~key~can~only~have~one~interface~declared~in~a~template.\\
    LaTeX~found~two~interfaces~for~'#1'.
  }
\msg_new:nnnn { xtemplate } { keytype-requires-argument }
  { The~key~type~'#1'~requires~an~argument~\msg_line_context:. }
  {
    You~should~have~put:\\
    \ \ <key-name>~:~#1~{~<argument>~} \\
    but~LaTeX~did~not~find~an~<argument>.
  }
\msg_new:nnnn { xtemplate } { invalid-keytype }
  { The~key~'#1'~is~missing~a~key-type~\msg_line_context:. }
  {
    \c_msg_coding_error_text_tl
    Each~key~in~a~template~requires~a~key-type,~given~in~the~form:\\
    \ \ <key>~:~<key-type>\\
    LaTeX~could~not~find~a~<key-type>~in~your~input.
  }
\msg_new:nnnn { xtemplate } { key-no-value }
  { The~key~'#1'~has~no~value~\msg_line_context:. }
  {
    \c_msg_coding_error_text_tl
    When~creating~an~instance~of~a~template~
    every~key~listed~must~include~a~value:\\
    \ \ <key>~=~<value>
  }
\msg_new:nnnn { xtemplate } { key-no-variable }
  { The~key~'#1'~requires~implementation~details~\msg_line_context:. }
  {
    \c_msg_coding_error_text_tl
    When~creating~template~code~using~\DeclareTemplateCode,~
    each~key~name~must~have~an~associated~implementation.\\
    This~should~be~given~after~a~'='~sign:~LaTeX~did~not~find~one.
  }
\msg_new:nnnn { xtemplate } { key-not-implemented }
  { Key~'#1'~has~no~implementation~\msg_line_context:. }
  {
    \c_msg_coding_error_text_tl
    The~definition~of~key~implementations~for~template~'#2'~
    of~object~type~'#3'~does~not~include~any~details~for~key~'#1'.\\
    The~key~was~declared~in~the~interface~definition,~
    and~so~an~implementation~is~required.
  }
\msg_new:nnnn { xtemplate } { missing-keytype }
```

```
  { The~key~'#1'~is missing~a~key-type~\msg_line_context:. }
  {
    \c_msg_coding_error_text_tl
    Key~interface~definitions~should~be~of~the~form\\
    \ \ #1~:~<key-type>\\
    but~LaTeX~could~not~find~a~<key-type>.
  }
\msg_new:nnnn { xtemplate } { no-template-code }
  {
    The~template~'#2'~of~type~'#1'~is~unknown~
    or~has~no~implementation.
  }
  {
    \c_msg_coding_error_text_tl
    There~is~no~code~available~for~the~template~name~given.\\
    This~should~be~given~using~\DeclareTemplateCode.
  }
\msg_new:nnnn { xtemplate } { object-type-mismatch }
  { Object~types~'#1'~and~'#2'~do~not~agree. }
  {
    You~are~trying~to~use~a~template~directly~with~\UseInstance
    (or~a~similar~function),~but~the~object~types~do~not~match.
  }
\msg_new:nnnn { xtemplate } { unknown-attribute }
  { The~template~attribute~'#1'~is~unknown. }
  {
    There~is~a~definition~in~the~current~template~reading\\
    \ \ \token_to_str:N \KeyValue {~#1~} \\
    but~there~is~no~key~called~'#1'.
  }
\msg_new:nnnn { xtemplate } { unknown-choice }
  { The~choice~'#2'~was~not~declared~for~key~'#1'. }
  {
    The~key~'#1'~takes~a~fixed~list~of~choices~
    and~this~list~does~not~include~'#2'.
  }
\msg_new:nnnn { xtemplate } { unknown-default-choice }
  { The~default~choice~'#2'~was~not~declared~for~key~'#1'. }
  {
    The~key~'#1'~takes~a~fixed~list~of~choices~
    and~this~list~does~not~include~'#2'.
  }
\msg_new:nnnn { xtemplate } { unknown-instance }
  { The~instance~'#2'~of~type~'#1'~is~unknown. }
  {
    You~have~asked~to~use~an~instance~'#2',~
    but~this~has~not~been~created.
  }
\msg_new:nnnn { xtemplate } { unknown-key }
  { Unknown~template~key~'#1'. }
  {
    \c_msg_coding_error_text_tl
    The~key~'#1'~was~not~declared~in~the~interface~
    for~the~current~template.
  }
\msg_new:nnnn { xtemplate } { unknown-keytype }
  { The~key-type~'#1'~is~unknown. }
  {
    \c_msg_coding_error_text_tl
    Valid~key-types~are:\\
    -~boolean;\\
    -~choice;\\
    -~code;\\
    -~commalist;\\
    -~function;\\
    -~instance;\\
    -~integer;\\
    -~length;\\
    -~muskip;\\
    -~real;\\
    -~skip;\\
    -~tokenlist.
  }
\msg_new:nnnn { xtemplate } { unknown-object-type }
  { The~object~type~'#1'~is~unknown. }
  {
    \c_msg_coding_error_text_tl
    An~object~type~needs~to~be~declared~with~\DeclareObjectType
    prior~to~using~it.
  }
\msg_new:nnnn { xtemplate } { unknown-template }
  { The~template~'#2'~of~type~'#1'~is~unknown. }
  {
    No~interface~has~been~declared~for~a~template~
    '#2'~of~object~type~'#1'.
  }
```

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

```
\msg_new:nnn { xtemplate } { declare-object-type }
  { Declaring~object~type~'#1'~taking~#2~argument(s)~\msg_line_context:. }
\msg_new:nnn { xtemplate } { declare-template-code }
  { Declaring~code~for~template~'#2'~of~object~type'#1'~\msg_line_context:. }
\msg_new:nnn { xtemplate } { declare-template-interface }
  {
    Declaring~interface~for~template~'#2'~of~object~type~'#1'~
    \msg_line_context:.
  }
\msg_new:nnn { xtemplate } { show-no-attribute }
  { The~template~'#2'~of~object~type~'#1'~has~no~#3 . }
\msg_new:nnn { xtemplate } { show-attribute }
```

```
  { The~template~'#2'~of~object~type~'#1'~has~#3 : }
\msg_new:nnn { xtemplate } { show-no-values }
  {
    The~ \tl_if_empty:nF {#2} {collection~} instance~'#3'~
    \tl_if_empty:nF {#2} { (from~collection~'#2')~ }
    of~object~type~'#1'~has~no~values.
  }
\msg_new:nnn { xtemplate } { show-values }
  {
    The~ \tl_if_empty:nF {#2} {collection~} instance~'#3'~
    \tl_if_empty:nF {#2} { (from~collection~'#2')~ }
    of~object~type~'#1'~
    \str_if_eq:nnF { \q_no_value } {#4} { (from~template~'#4')~ }
    has~values:
  }
```

11.16 User functions

The user functions provided by xtemplate are pretty much direct copies of internal ones. However, by sticking to the xparse approach only the appropriate arguments are long.

\DeclareObjectType
\DeclareTemplateInterface
\DeclareTemplateCode
\DeclareRestrictedTemplate
\EditTemplateDefaults
\DeclareInstance
\DeclareCollectionInstance
\EditInstance
\EditCollectionInstance
\UseTemplate
\UseInstance
\UseCollection

All simple translations, with the appropriate long/short argument filtering.

```
\cs_new_protected_nopar:Npn \DeclareObjectType #1#2
  { \__xtemplate_declare_object_type:nn {#1} {#2} }
\cs_new_protected:Npn \DeclareTemplateInterface #1#2#3#4
  { \__xtemplate_declare_template_keys:nnnn {#1} {#2} {#3} {#4} }
\cs_new_protected:Npn \DeclareTemplateCode #1#2#3#4#5
  { \__xtemplate_declare_template_code:nnnnn {#1} {#2} {#3} {#4} {#5} }
\cs_new_protected:Npn \DeclareRestrictedTemplate #1#2#3#4
  { \__xtemplate_declare_restricted:nnnn {#1} {#2} {#3} {#4} }
\cs_new_protected:Npn \DeclareInstance #1#2#3#4
  { \__xtemplate_declare_instance:nnnnn {#1} {#3} { } {#2} {#4} }
\cs_new_protected:Npn \DeclareCollectionInstance #1#2#3#4#5
  { \__xtemplate_declare_instance:nnnnn {#2} {#4} {#1} {#3} {#5} }
\cs_new_protected:Npn \EditTemplateDefaults #1#2#3
  { \__xtemplate_edit_defaults:nnn {#1} {#2} {#3} }
\cs_new_protected:Npn \EditInstance #1#2#3
  { \__xtemplate_edit_instance:nnnn {#1} { } {#2} {#3} }
\cs_new_protected:Npn \EditCollectionInstance #1#2#3#4
  { \__xtemplate_edit_instance:nnnn {#2} {#1} {#3} {#4} }
\cs_new_protected_nopar:Npn \UseTemplate #1#2#3
  { \__xtemplate_use_template:nnn {#1} {#2} {#3} }
\cs_new_protected_nopar:Npn \UseInstance #1#2
  { \__xtemplate_use_instance:nn {#1} {#2} }
\cs_new_protected_nopar:Npn \UseCollection #1#2
  { \__xtemplate_use_collection:nn {#1} {#2} }
```

(*End definition for* `\DeclareObjectType` *This function is documented on page* **??**.)

\ShowTemplateCode
\ShowTemplateDefaults
\ShowTemplateInterface
\ShowTemplateVariables
\ShowInstanceValues
\ShowCollectionInstanceValues

The show functions are again just translation.

```
\cs_new_protected_nopar:Npn \ShowTemplateCode #1#2
  { \__xtemplate_show_code:nn {#1} {#2} }
\cs_new_protected_nopar:Npn \ShowTemplateDefaults #1#2
  { \__xtemplate_show_defaults:nn {#1} {#2} }
\cs_new_protected_nopar:Npn \ShowTemplateInterface #1#2
  { \__xtemplate_show_keytypes:nn {#1} {#2} }
\cs_new_protected_nopar:Npn \ShowTemplateVariables #1#2
  { \__xtemplate_show_vars:nn {#1} {#2} }
\cs_new_protected_nopar:Npn \ShowInstanceValues #1#2
  { \__xtemplate_show_values:nnn {#1} { } {#2} }
\cs_new_protected_nopar:Npn \ShowCollectionInstanceValues #1#2#3
  { \__xtemplate_show_values:nnn {#1} {#2} {#3} }
```

(*End definition for* `\ShowTemplateCode` *This function is documented on page 9.*)

`\IfInstanceExistTF` More direct translation: only the base instance is checked for.

```
\cs_new_nopar:Npn \IfInstanceExistTF #1#2
  { \__xtemplate_if_instance_exist:nnnTF {#1} { } {#2} }
\cs_new_nopar:Npn \IfInstanceExistT #1#2
  { \__xtemplate_if_instance_exist:nnnT {#1} { } {#2} }
\cs_new_nopar:Npn \IfInstanceExistF #1#2
  { \__xtemplate_if_instance_exist:nnnF {#1} { } {#2} }
```

(*End definition for* `\IfInstanceExistTF` *This function is documented on page* ??.)

`\EvaluateNow`
`\KeyValue`
These are both do nothing functions. Both simply dump their arguments when executed: this should not happen with `\KeyValue`.

```
\cs_new_protected:Npn \EvaluateNow #1 {#1}
\cs_new_protected:Npn \KeyValue #1 {#1}
```

(*End definition for* `\EvaluateNow` *This function is documented on page 4.*)

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

```
\cs_new_protected_nopar:Npn \AssignTemplateKeys
  { \__xtemplate_assignments_pop: }
```

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

```
\cs_new_eq:NN \ShowTemplateKeytypes \ShowTemplateInterface
⟨/package⟩
```

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.