The xtemplate package*

The LaTeX3 Project†

2010/10/03

1 Introduction

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

1. Authoring of the text, with mark-up
2. Document layout design
3. Implementation (with 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 2_ε classes look somewhat dated now in terms of their visual design, their typography is generally sound. (Barring the occasional minor faults.)

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

- Creating a new version of the implementation code of the class and editing it.
- Loading one of the many packages to customise certain elements of the standard classes.

*This file has version number 2063, last revised 2010/10/03.

†Frank Mittelbach, Denys Duchier, Chris Rowley, Rainer Schöpf, Johannes Braams, Michael Downes, David Carlisle, Alan Jeffrey, Morten Høgholm, Thomas Lotze, Javier Bezos, Will Robertson, Joseph Wright

- 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.

2 What is a document?

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

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

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

There are three distinct layers in the definition of 'a document' at this level:

1. Semantic elements such as the ideas of sections and lists.
2. A set of design solutions for representing these elements visually.
3. Specific variations for these designs that represent the elements in the document.

In the parlance of the xtemplate package, we call these object types, templates, and instances, and they are discussed below in sections 3.1, 3.2, and 3.4, respectively.

3 Objects, templates, and instances

By formally declaring our document 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. The xtemplate package provides the tools to do this.

3.1 Object types

An 'object type' (or 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 {`⟨*name*⟩`} {`⟨*Nargs*⟩`}`

This function defines an *object type*, where ⟨*name*⟩ is the name of the object type and ⟨*Nargs*⟩ is the number of arguments an instance of this type should take. For example,

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

Note that object types are global entities: `\DeclareObjectType` will apply outside of any TEX grouping in force when it is called.

3.2 Templates

A *template* is a generalised design solution for representing the information of a specified *object type*. Templates that do the same thing — e.g., two completely different ways of printing a chapter heading — 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`

`\DeclareTemplateInterface {`⟨*object type*⟩`} {`⟨*template*⟩`} {`⟨*Nargs*⟩`}`
`{`
⟨*name of key* 1⟩ `:` ⟨*key type* 1⟩ `,`
⟨*name of key* 2⟩ `:` ⟨*key type* 2⟩ `=` ⟨*optional default*⟩ `,`
...
`}`

The ⟨*name of keys*⟩ can be any string of ASCII characters (with the exception of `:`, `=`

and , as they are part of the syntax); we recommend only using lower case letters and dashes, however. Note that spaces in key names are ignored, so that key names can be spaced out for ease of reading without affecting the recognition of keys inside and outside of code blocks.

The ⟨*key types*⟩ define what sort of input the key accepts, such as 'boolean', 'integer', and so on. The complete list of possible ⟨*key types*⟩ is shown in Table 1.

Like objects, templates are global entities: both `\DeclareTemplateInterface` `\DeclareTemplateCode` will apply outside of any TEX grouping in force when it is called.

`\DeclareTemplateCode`

`\DeclareTemplateCode` `{`⟨*object type*⟩`}` `{`⟨*template*⟩`}` `{`⟨*Nargs*⟩`}`
`{`
  ⟨*name of key*$_1$⟩ `=` ⟨*internal variable or code*$_1$⟩ `,`
  ⟨*name of key*$_2$⟩ `=` ⟨*internal variable or code*$_2$⟩ `,`
  ...
`}{`
  ⟨*implementation code*⟩
  `\AssignTemplateKeys`
  ⟨*more implementation code*⟩
`}`

After the keys have been declared with `\DeclareTemplateInterface`, the implementation binds each ⟨*name of key*⟩ with an ⟨*internal variable*⟩ (for key types such as 'integer', 'length', 'tokenlist', etc.)[1] or with a certain ⟨*code*⟩ fragment to execute, which will be described below.

Assignments to variables which should be made globally are indicated by adding the word `global` before the variable name:

⟨*name of key*$_1$⟩ `=` ⟨*internal variable*$_1$⟩ `,`
⟨*name of key*$_2$⟩ `= global` ⟨*internal variable*$_2$⟩ `,`

The key types `choices` and `code` do not take variable bindings; instead, fragments of code are defined which are executed instead. The complete list of bindings taken by different key types is shown in Table 2. The `choices` key type is explained fully in subsection 3.3 below.

`\AssignTemplateKeys`

The final argument of `\DeclareTemplateCode` contains the ⟨*implementation code*⟩ for the template design, taking arguments `#1`, `#2`, etc. according to the number of arguments allowed, ⟨*Nargs*⟩. `\AssignTemplateKeys` must be executed in order to assign variables and perform code executions according to the keys set.

[1] It is possible, if you wish, to use the same variable for multiple keys; this allows 'key synonyms' to be defined such as `color` and `colour` which can perform the same function in the template implementation.

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 of arbitrary items
`function` N	A function definition with N arguments (N from 0 to 9)
`instance {`⟨*name*⟩`}`	An instance of type ⟨*name*⟩
`integer`	An integer expression (e.g., `(1 + 5)/2`)
`length`	A dimension expression (e.g., `3pt + 2cm`)
`skip`	A dimension expression with glue (e.g., `3pt plus 2pt minus 1pt`)
`tokenlist`	A 'token list' input; any text or commands

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

Key Type		Description of binding
`boolean`	⋆	Boolean variable; e.g., `\l_tmpa_bool`
`choice`		`{` ⟨*choice*$_1$⟩ `=` ⟨*code*$_1$⟩ `,` ⟨*choice*$_2$⟩ `=` ⟨*code*$_2$⟩ `, ...}`
`code`		⟨*code*⟩ ; use `#1` as the input to the key
`commalist`	⋆	Comma-list variable; e.g., `\l_tmpa_clist`
`function`	⋆	Function w/ N arguments; e.g., `\use_i:nn`
`instance`	⋆	An instance variable; e.g., `\g_foo_instance`
`integer`	⋆	Integer variable; e.g., `\l_tmpa_int`
`length`	⋆	Dimension variable; e.g., `\l_tmpa_dim`
`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`. Starred entries may be prefixed with the keyword `global` to make a global assignment.

3.3 Multiple choices

The `choice` keytype 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 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.

3.4 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 12pt italic with a 10pt skip before and a 12pt 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*⟩`}`
The name of the instance being declared is ⟨*instance*⟩, with ⟨*parameters*⟩ the keyval input to set some or all of the ⟨*template*⟩ keys to specific values.

Here is a hypothetical example, where `sectioning` might be an object to be used for document subdivisions, `section-num` an instance referring to a 'numbered section', and `basic` a template for `sectioning` that performs just the basic layout, say:

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

3.5 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*⟩

It will take as many arguments as were defined for the object type.

Use xparse to declare the document commands in terms of instances. Another hypothetical example:

```
\DeclareDocumentCommand\section{ som }{
  \IfBooleanTF #1
  {
    \UseInstance{sectioning}{section-nonum}{#2}{#3}
  }
  {
    \UseInstance{sectioning}{section-num}{#2}{#3}
  }
}
```

`\UseTemplate`

`\UseTemplate {`⟨*object type*⟩`} {`⟨*template*⟩`} {`⟨*settings*⟩`}` ⟨*arguments*⟩
There are occasions where creating an instance of a template does not make sense, as it will only be used once. In this case, templates can be used directly, with the key settings given as an argument to the `\UseTemplate` function. This will also work when giving an argument to a key which needs an instance. For example, if we have an key `instance-key` which expects an instance of `object2`, then we can either declare an instance:

```
\DeclareInstance {object2} {template2} {temp-instance} {
  <settings>
}
\DeclareInstance {object} {template} {instance} {
  instance-key = temp-instance
}
```

or use the template directly:

```
\DeclareInstance {object} {template} {instance} {
  instance-key = \UseTemplate {object2} {template2} {<settings>}
}
```

Which is the best approach will depend on the exact nature of the situation.

3.6 Summaries

For the document designer:

- The class will define which object types are used in a document.
- The class will define user commands that contain the required instances that the document must use.

- Having knowledge of a variety of suitable templates, for each required instance a template can be selected and instantiated based on the parameters defined by `\DeclareTemplateInterface`.

For the class programmer:

- Define the different object types of document elements: what the semantics are and what information is required.
- Create document commands to call instances that fulfil the needs of the object types.
- Implement the required templates to produce typeset implementations of the document elements and instantiate them with the appropriate names.

4 Instances in different contexts

We may wish the behaviour of an instance to change as it is used in varying contexts. For example, in the frontmatter of a document, section numbering is different. Semantics are the same, but the typesetting changes. But we want to use the same user commands, and hence the same instance names.

Collections allow us to define multiple instances that we can switch between. Collections are activated with `\UseCollection`.

At present, it is not clear whether collections fully address the issues they target. They should therefore be regarded as highly experimental, and may be changed or withdrawn in the future if it appears that they do not work well enough!

`\DeclareCollectionInstance`

`\DeclareCollectionInstance` {⟨*collection*⟩} {⟨*object type*⟩} {⟨*instance*⟩} {⟨*template*⟩} {⟨*parameters*⟩}

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

The instance declared will override another instance of the same name when the collection is active. Note that a collection instance can only be declared if the *original* instance already exists.

An example might be:

```
\UseCollection{sectioning}{frontmatter}
```

```
\section{Nomenclature}
...
\UseCollection{sectioning}{default}
\section{Introduction}
```

In both cases, the same instance (perhaps '`section-num`') is being used inside the `\section`. But `\DeclareCollectionInstance` will have been used for the '`frontmatter`' and override the instance that is used in the default case.

5 Bits 'n' pieces

5.1 Does an instance exist?

`\IfInstanceExist`*TF*

`\IfInstanceExistTF {`⟨*object type*⟩`} {`⟨*instance*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfInstanceExistT {`⟨*object type*⟩`} {`⟨*instance*⟩`} {`⟨*true code*⟩`}`
`\IfInstanceExistF {`⟨*object type*⟩`} {`⟨*instance*⟩`} {`⟨*false code*⟩`}`
Test if ⟨*instance*⟩ has been declared. This is useful when the use of an instance depends on some global variable, such as the current font selection. Designers or users can then implement specific designs for exact situations rather than relying on blanket parameter redefinitions. See xfrac for a good example of this.

5.2 Changing the defaults of a template's keys

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*⟩`}`

This command only takes effect for instances that have not yet been declared. Use `\EditInstance` if you wish to change an instance that already exists.

5.3 Small changes to an instance

When a designer creates an instance but the user wishes to slightly tweak it, it is convenient to not have to reset all of the (possibly many) parameters defining that instance and only override the specific parameter that should be changed.

`\EditInstance`
`\EditCollectionInstance`

`\EditInstance {`⟨*object type*⟩`} {`⟨*instance*⟩`} {`⟨*new parameters*⟩`}`
`\EditCollectionInstance {`⟨*object type*⟩`} {`⟨*collection*⟩`} {`⟨*instance*⟩`}`
`{`⟨*template*⟩`} {`⟨*new parameters*⟩`}`

These functions change the key settings of an instance of an object type. If the instance was derived from a template, this information is used to find the correct keys to use for the editing process. It may be convenient to use `\ShowInstanceValues` to inspect the values used to set the keys originally.

5.4 Parameters evaluated now

`\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!

5.5 Setting one key to the value of another

It is often useful to use the value of one key as the default for another.

`\KeyValue` `\KeyValue {`⟨*key name*⟩`}`

This command is used as the argument to an instance key; it will set that key to the value of ⟨*key name*⟩ each time the instance is executed at run-time. Using `\KeyValue` means that the designer does not need to know how a particular key has been implemented.

5.6 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` creates a derived version of a template for which certain parameters are frozen as specified but the remaining parameters are available to be set as usual in an instance declaration.

`\DeclareRestrictedTemplate`

`\DeclareRestrictedTemplate {`⟨*object type*⟩`} {`⟨*parent*⟩`} {`⟨*new template*⟩`} {`⟨*frozen parameters*⟩`}`
Defines ⟨*new template*⟩ based on template ⟨*parent*⟩ (of certain ⟨*object type*⟩) with certain keys set and frozen as defined in ⟨*keyvals*⟩.

6 Getting information about templates and instances

`\ShowTemplateCode`
`\ShowTemplateDefaults`
`\ShowTemplateKeytypes`
`\ShowTemplateVariables`

`\ShowTemplateCode {`⟨*object type*⟩`} {`⟨*template*⟩`}`
These functions pause the typesetting and display in the console the various pieces of information for a template.

`\ShowInstanceValues`
`\ShowCollectionInstanceValues`

`\ShowInstanceValues {`⟨*object type*⟩`} {`⟨*instance*⟩`}`
`\ShowCollectionInstanceValues {`⟨*object type*⟩`} {`⟨*collection*⟩`} {`⟨*instance*⟩`}`
These functions pause the typesetting and display in the console information about an instance or a collection instance.

Note that xtemplate uses various special key names internally. These all contain a space when stored (which normal keys do not: spaces are removed). The same applies to choices: these are stored internally as ⟨***key***⟩ ⟨***choice***⟩. These will show up when using the `\Show...` functions. The design means that there is no danger of a clash between user keys and internal keys. Also, standard keys are stored with all letters detokenized, whereas the special keys use letters with category code 11 (letter), again to avoid any issues.

7 Examples

(Nothing here yet.)

8 Code documentation

8.1 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_restrict_root_tl`
`\c_xtemplate_values_root_tl`
`\c_xtemplate_vars_root_tl`
A number of pieces of code and lists of properties have to be stored for templates and instances. The various csname roots are set up as token lists to avoid use of the literal text in the code.

`\c_xtemplate_key_order_tl` The order keys are declared in must be stored (as property lists have no 'order'). The special property used is named here.

`\c_xtemplate_keytypes_arg_clist` Some keytypes (such as `instance`) need additional information, given as an argument. The list of keytypes that need this extra data is set up here, for later use when splitting things.

`\g_xtemplate_object_type_prop` For tracking which object types have been declared, and the number of arguments each requires.

`\l_xtemplate_assignments_tl` This token list variable is used in two places. First, it is where the list of assignments for an instance is constructed during `\DeclareInstance`. Second, it is where these are copied to to allow `\AssignTemplateKeys` to work correctly.

`\l_xtemplate_collection_tl` The name of the current instance collection active. If no collection is in use, this will simply be empty.

`\l_xtemplate_collections_prop` Records the collection in force for each object type.

`\l_xtemplate_default_tl`
`\l_xtemplate_key_name_tl`
`\l_xtemplate_keytype_tl`
`\l_xtemplate_keytype_arg_tl`
`\l_xtemplate_value_tl`
`\l_xtemplate_var_tl`
When processing keys, various properties for the current key need to be available. These are copied from the property list to appropriately named token lists, and back again, as needed.

`\l_xtemplate_error_bool` Used to indicate an error when parsing a key list, so that further processing can be abandoned.

`\l_xtemplate_global_bool` When actually assigning data to variables, a check is made to see if this should be global. The flag here is used to indicate this.

`\l_xtemplate_key_seq` The order in which keys are defined is stored here for later recovery and use. It is transferred into the property list for the template when the template is saved.

`\l_xtemplate_restrict_bool` Flag used when editing templates, so that simple editing and restricting can share the same underlying editing method.

`\l_xtemplate_restricted_clist`
`\l_xtemplate_keytypes_prop`
`\l_xtemplate_values_prop`
`\l_xtemplate_vars_prop`
To avoid needing to refer to the data about a template or instance by csname in a large number of locations, the data is copied to these scratch variables and back again for processing. This makes the code easier to follow.

`\l_xtemplate_tmp_clist`
`\l_xtemplate_tmp_dim`
`\l_xtemplate_tmp_int`
`\l_xtemplate_tmp_skip`
`\l_xtemplate_tmp_tl`
Used when carrying out assignments, as the pre-processing can take place here before passing data through to the storage area defined by the implementation part of a template. The token list is also used for general scratch purposes by xtemplate.

`\l_xtemplate_restrict_bool` Flag used when editing templates, so that simple editing and restricting can share the same underlying editing method.

8.2 Execute or error functions

These all either execute code (if the tests are true) or issue errors (if the test fails).

`\xtemplate_execute_if_arg_agree:nnT` `\xtemplate_execute_if_arg_agree:nnT {⟨type⟩} {⟨num⟩} {⟨true code⟩}`

Tests if the number of arguments required by ⟨*type*⟩ is equal to ⟨*num*⟩, then executes either ⟨*true code*⟩ or generates an error as appropriate.

`\xtemplate_execute_if_code_exist:nnT` `\xtemplate_execute_if_code_exist:nnT {⟨type⟩} {⟨templat`
`{⟨true code⟩}`

Tests if ⟨*template*⟩ of ⟨*type*⟩ has been defined (i.e., the code has been created for an implementation), then executes either ⟨*true code*⟩ or generates an error as appropriate.

`\xtemplate_execute_if_keytype_exist:nT`
`\xtemplate_execute_if_keytype_exist:VT` `\xtemplate_execute_if_keytype_exist:nT {⟨keytype⟩}`
`{⟨true code⟩}`

Tests if ⟨*keytype*⟩ is a known keytype, then executes either ⟨*true code*⟩ or generates an error as appropriate.

`\xtemplate_execute_if_type_exist:nT` `\xtemplate_if_type_exist:nT {⟨type⟩} {⟨true code⟩}`

Tests if template ⟨*type*⟩ has been created, then executes either ⟨*true code*⟩ or generates an error as appropriate.

`\xtemplate_execute_if_keys_exist:nnT` `\xtemplate_if_keys_exist:nnT {⟨type⟩} {⟨template⟩}`
`{⟨true code⟩}`

Tests if keys for ⟨*template*⟩ of ⟨*type*⟩ have been declared (but not necessarily given an implementation), , then executes either ⟨*true code*⟩ or generates an error as appropriate.

8.3 Utility functions

`\xtemplate_if_key_value:nT ⋆`
`\xtemplate_if_key_value:VT ⋆` `\xtemplate_if_key_value:nT {⟨tokens⟩} {⟨true code⟩}`

Tests if the first token in ⟨*tokens*⟩ is `\KeyValue`.

`\xtemplate_if_eval_now:nTF ⋆` `\xtemplate_if_eval_now:nTF {⟨tokens⟩}`
`{⟨true code⟩} {⟨false code⟩}`

Tests if the first token in ⟨*tokens*⟩ is a marker for evaluating now (`\EvaluateNow`).

`\xtemplate_if_instance_exist:nnn`*TF* ⋆ `\xtemplate_if_instance_exist:nnnTF {⟨type⟩}`
`{⟨collection⟩} {⟨instance⟩} {⟨true code⟩}`
`{⟨false code⟩}`

Tests if ⟨*instance*⟩ of ⟨*type*⟩ exists for the ⟨*collection*⟩ given.

`\xtemplate_if_use_template:nTF ⋆` `\xtemplate_if_use_template:nTF {⟨assignment⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if assignment begins with `\UseTemplate`.

`\xtemplate_store_defaults:n`
`\xtemplate_store_keytypes:n`
`\xtemplate_store_restrictions:n`
`\xtemplate_store_values:n`
`\xtemplate_store_vars:n`
`\xtemplate_store_defaults:n {⟨full name⟩}`

These functions copy information about the current template or instance from the scratch variables to those for storing the information. The ⟨*full name*⟩ of the instance or template is needed: this includes the ⟨*type*⟩ and ⟨*collection*⟩ (if applicable).

`\xtemplate_recover_defaults:n`
`\xtemplate_recover_keytypes:n`
`\xtemplate_recover_restrictions:n`
`\xtemplate_recover_values:n`
`\xtemplate_recover_vars:n`
`\xtemplate_recover_defaults:n {⟨full name⟩}`

The reverse of the `store` functions, these functions copy data from the storage areas to the scratch variables for use in the module. Again, the ⟨*full name*⟩ is needed, including the ⟨*type*⟩.

8.4 Creating object types

`\xtemplate_declare_object_type:nn` `\xtemplate_declare_object_type:nn {⟨type⟩} {⟨num⟩}`

Declares ⟨*type*⟩ of object, to accept ⟨*num*⟩ arguments.

8.5 Declaring template keys

`\xtemplate_declare_template_keys:nnnn` `\xtemplate_declare_template_keys:nnnn {⟨type⟩} {⟨template⟩} {⟨num⟩} {⟨keyvals⟩}`

Declares ⟨*template*⟩ of ⟨*type*⟩, and accepting ⟨*num*⟩ arguments, with key types and default values defined by ⟨*keyvals*⟩.

\xtemplate_parse_keys_elt:n
\xtemplate_parse_keys_elt:nn \xtemplate_parse_keys_elt:nn {⟨*key*⟩} {⟨*value*⟩}

Functions used to process each key–value pair when declaring keys from ⟨*keyvals*⟩.

\xtemplate_split_keytype:n \xtemplate_split_keytype:n {⟨*key*⟩}

Splits a ⟨*key*⟩ into a key name (stored as \l_xtemplate_key_tl) and a keytype (stored as \l_xtemplate_keytype_tl).

\xtemplate_split_keytype_arg:n
\xtemplate_split_keytype_arg:V \xtemplate_split_keytype_arg:n {⟨*keytype*⟩}

Splits a ⟨*keytype*⟩ into the type itself and any optional qualifying text. The results are stored in \l_xtemplate_keytype_tl and \l_xtemplate_keytype_arg_tl.

8.6 Storing defaults and values

\xtemplate_store_value_boolean:n
\xtemplate_store_value_choice:n
\xtemplate_store_value_choice:V
\xtemplate_store_value_code:n
\xtemplate_store_value_commalist:n
\xtemplate_store_value_function:n
\xtemplate_store_value_function:n
\xtemplate_store_value_instance:n
\xtemplate_store_value_tokenlist:n
\xtemplate_store_value_integer:n
\xtemplate_store_value_length:n
\xtemplate_store_value_skip:n \xtemplate_store_value_boolean:n {⟨*value*⟩}

Store values of the given keytype for later assignment to variables. For the numeric and Boolean data types, the value is evaluated at this stage unless \DelayEvaluation or \KeyValue are used in the ⟨*value*⟩.

\xtemplate_store_value_choice_name:n \xtemplate_store_value_choice_name:n {⟨*value*⟩}

Stores the name of a choice for a multiple choice key, which will be turned into an implementation when code is available.

8.7 Implementing templates

`\xtemplate_declare_template_code:nnnnn` `\xtemplate_declare_template_code:nnnnn {⟨type⟩} {⟨template⟩} {⟨num⟩} {⟨keyvals⟩} {⟨code⟩}`

Declares implementation of ⟨*template*⟩ of ⟨*type*⟩, and accepting ⟨*num*⟩ arguments, with keys implemented as listed in ⟨*keyvals*⟩ and with ⟨*code*⟩ to be executed when the ⟨*template*⟩ is used.

`\xtemplate_store_key_implementation:nnn` `\xtemplate_store_key_implementation:nnn {⟨type⟩} {⟨template⟩} {⟨keyvals⟩}`

Stores the implementation for keys as specified in ⟨*keyvals*⟩ for a ⟨*template*⟩ of ⟨*type*⟩.

`\xtemplate_parse_vars_elt:n` `\xtemplate_parse_vars_elt:nn` `\xtemplate_parse_vars_elt:nn {⟨key⟩} {⟨variable⟩}`

Used by the key–value parser to assign a ⟨*variable*⟩ for each ⟨*key*⟩ listed.

`\xtemplate_store_key_implementation:nnn` `\xtemplate_store_key_implementation:nnn {⟨type⟩} {⟨template⟩} {⟨keyvals⟩}`

Stores the implementation for keys as specified in ⟨*keyvals*⟩ for a ⟨*template*⟩ of ⟨*type*⟩.

`\xtemplate_implement_choices:n` `\xtemplate_implement_choices:n key-value list`

Master function for turning ⟨*key-value lsit*⟩ into a set of choices.

`\xtemplate_implement_choice_elt:n` `\xtemplate_implement_choice_elt:nn` `\xtemplate_implement_choice_elt:nn {⟨choice⟩} {⟨code⟩}`

Used by the key–value parser to convert a key–value list of choices and code into working multiple choice values.

8.8 Modifying templates

`\xtemplate_declare_restricted:nnnn` `\xtemplate_declare_restricted:nnnn {⟨type⟩} {⟨parent⟩} {⟨restricted⟩} {⟨keyvals⟩}`

Creates ⟨*restricted*⟩ template of ⟨*type*⟩ based on ⟨*parent*⟩ by fixing values as listed in ⟨*keyvals*⟩.

`\xtemplate_edit_defaults:nnn` `\xtemplate_edit_defaults:nnn {⟨type⟩} {⟨template⟩} {⟨keyvals⟩}`

Modifies the default values for ⟨*template*⟩ of ⟨*type*⟩ as instructed in ⟨*keyvals*⟩.

`\xtemplate_parse_values:nn` `\xtemplate_parse_values:nn {⟨name⟩} {⟨keyvals⟩}`

Parses ⟨*keyvals*⟩ for full ⟨*name*⟩, finding the value for each key and storing it for later assignment.

`\xtemplate_parse_values_elt:n`
`\xtemplate_parse_values_elt:nn` `\xtemplate_parse_values_elt:nn {⟨key⟩} {⟨variable⟩}`

Used by the key–value parser to find ⟨*value*⟩ to assing to implementation of ⟨*key*⟩.

`\xtemplate_set_template_eq:nn` `\xtemplate_set_template_eq:nn {⟨copy⟩} {⟨parent⟩}`

Copies all of ⟨*parent*⟩ template to the ⟨*copy*⟩, where both are full names (i.e., a template plus type).

8.9 Creating instances

`\xtemplate_declare_instance:nnnnn` `\xtemplate_declare_instance:nnnnn {⟨type⟩} {⟨template⟩} {⟨collection⟩} {⟨instance⟩} {⟨keyvals⟩}`

Declares an ⟨*instance*⟩ (within ⟨*collection*⟩) of ⟨*template*⟩ of ⟨*type*⟩, using ⟨*keyvals*⟩ to define the instance.

`\xtemplate_edit_instance:nnnn` `\xtemplate_declare_instance:nnnn {⟨type⟩} {⟨collection⟩} {⟨instance⟩} {⟨keyvals⟩}`

Modifies an ⟨*instance*⟩ (within ⟨*collection*⟩) of ⟨*type*⟩, using ⟨*keyvals*⟩ to modify the instance.

`\xtemplate_convert_to_assignments:` `\xtemplate_convert_to_assignments:`

Converts the contents of the various scratch property lists into a list of variable assignments in `\l_xtemplate_assignments_tl`.

`\xtemplate_find_global:` `\xtemplate_find_global:`

Checks in `\l_xtemplate_var_tl` for the special text `global`, which is removed from the variable is found. The flag `\l_xtemplate_global_bool` is then set as appropriate.

8.10 Converting values to assignments

\xtemplate_assign_boolean:
\xtemplate_assign_choice:
\xtemplate_assign_code:
\xtemplate_assign_code:n
\xtemplate_assign_commalist:
\xtemplate_assign_function:
\xtemplate_assign_instance:
\xtemplate_assign_integer:
\xtemplate_assign_length:
\xtemplate_assign_skip:
\xtemplate_assign_tokenlist:
`\xtemplate_assign_boolean:`

Convert the given ⟨*keytype*⟩ of ⟨*key*⟩ into an assignment to a ⟨*variable*⟩.

`\xtemplate_assign_variable:N` `\xtemplate_assign_variable:N` ⟨*function*⟩

Convert the current contents of `\l_xtemplate_value_tl` into an assignment using ⟨*function*⟩ to the variable named in `\l_xtemplate_var_tl`.

`\xtemplate_key_to_value:` `\xtemplate_key_to_value:`

Converts an attribute named using `\KeyValue` into the value of the underlying implementation variable.

8.11 Using instances

`\xtemplate_use_instance:nn` `\xtemplate_use_instance:nn {`⟨*type*⟩`} {`⟨*instance*⟩`}`

Executes code stored for ⟨*instance*⟩ of ⟨*type*⟩, taking account of any active collection.

`\xtemplate_use_template:nnn` `\xtemplate_use_template:nnn {`⟨*type*⟩`} {`⟨*template*⟩`} {`⟨*settings*⟩`}`

Executes code stored for ⟨*template*⟩ of ⟨*type*⟩ using ⟨*settings*⟩.

`\xtemplate_use_collection:nn` `\xtemplate_use_collection:nn {`⟨*type*⟩`} {`⟨*collection*⟩`}`

Activates ⟨*collection*⟩ for instances of `{`⟨*type*⟩`}`.

`\xtemplate_get_collection:n` `\xtemplate_get_collection:n {⟨type⟩}`

Sets `\l_xtemplate_collection_tl` to the name of the collection in force for templates of ⟨*type*⟩.

`\xtemplate_assignments_pop:` `\xtemplate_assignments_pop:`

Pops `\l_xtemplate_assignment_tl`, and therefore executes the assignements stored there.

`\xtemplate_assignments_push:n` `\xtemplate_assignments_push:n {⟨assignments⟩}`

Pushes ⟨*assignments*⟩ to `\l_xtemplate_assignment_tl` for later execution.

8.12 Showing details

`\xtemplate_show_code:nn` `\xtemplate_show_code:nn {⟨type⟩} {⟨template⟩}`

Shows code associated with ⟨*template*⟩ of ⟨*type*⟩.

`\xtemplate_show_code:nn` `\xtemplate_show_code:nn {⟨type⟩} {⟨template⟩}`

Shows code associated with ⟨*template*⟩ of ⟨*type*⟩.

`\xtemplate_show_defaults:nn` `\xtemplate_show_default:nn {⟨type⟩} {⟨template⟩}`

Shows default values associated with ⟨*template*⟩ of ⟨*type*⟩.

`\xtemplate_show_keytypes:nn` `\xtemplate_show_keytypes:nn {⟨type⟩} {⟨template⟩}`

Shows key types associated with ⟨*template*⟩ of ⟨*type*⟩.

`\xtemplate_show_values:nnn` `\xtemplate_show_code:nnn {⟨type⟩} {⟨collection⟩} {⟨instance⟩}`

Shows values associated with ⟨*instance*⟩ of ⟨*type*⟩ within ⟨*collection*⟩.

`\xtemplate_show_vars:nn` `\xtemplate_show_vars:nn {⟨type⟩} {⟨template⟩}`

Shows variables associated with ⟨*template*⟩ of ⟨*type*⟩.

9 Implementation

xtemplate only needs expl3; in format mode, this can be skipped.

```
⟨*package⟩
\ProvidesExplPackage
  {\filename}{\filedate}{\fileversion}{\filedescription}
\RequirePackage{expl3}
⟨/package⟩
⟨*initex | package⟩
```

9.0.1 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_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      { xtemplate_code >  }
\tl_new:Nn \c_xtemplate_defaults_root_tl  { xtemplate_defaults >  }
\tl_new:Nn \c_xtemplate_instances_root_tl { xtemplate_instances >  }
\tl_new:Nn \c_xtemplate_keytypes_root_tl  { xtemplate_keytypes >  }
\tl_new:Nn \c_xtemplate_restrict_root_tl  { xtemplate_restrict >  }
\tl_new:Nn \c_xtemplate_values_root_tl    { xtemplate_values >  }
\tl_new:Nn \c_xtemplate_vars_root_tl      { xtemplate_vars >  }
```

`\c_xtemplate_key_order_tl`

A special property name, used to store the order keys are defined in.

```
\tl_new:Nn \c_xtemplate_key_order_tl { key~order }
```

`\c_xtemplate_keytypes_arg_clist`

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

```
\clist_new:N \c_xtemplate_keytypes_arg_clist
\clist_put_right:Nn  \c_xtemplate_keytypes_arg_clist { choice }
\clist_put_right:Nn  \c_xtemplate_keytypes_arg_clist { function }
\clist_put_right:Nn  \c_xtemplate_keytypes_arg_clist { instance }
```

`\g_xtemplate_object_type_prop`

For storing types and the associated number of arguments.

```
\prop_new:N \g_xtemplate_object_type_prop
```

`\l_xtemplate_assignments_tl`

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

```
\tl_new:N \l_xtemplate_assignments_tl
```

`\l_xtemplate_collection_tl`

The current instance collection name is stored here.

```
\tl_new:N \l_xtemplate_collection_tl
```

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

```
\prop_new:N \l_xtemplate_collections_prop
```

`\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
```

`\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_key_seq` The order that keys are declared needs to be know, so that they can be set in the same way. As property lists are not ordered data types, a separate list needs to be kept which *is* ordered. This will then be stored in the property list.

```
\seq_new:N \l_xtemplate_key_seq
```

`\l_xtemplate_keytypes_prop`
`\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
\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_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
\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
```

9.1 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.

mplate_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: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:nTF { #2 = \l_xtemplate_tmp_tl }
     {#3}
     {
       \msg_kernel_error:nnxxx { xtemplate }
         { argument-number-mismatch } {#1} { \l_xtemplate_tmp_tl } {#2}
     }
}
```

plate_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:Npn \xtemplate_execute_if_code_exist:nnT #1#2#3 {
  \cs_if_exist:cTF { \c_xtemplate_code_root_tl #1 / #2 :w }
    {#3}
    {
      \msg_kernel_error:nnxx { xtemplate } { no-template-code }
        {#1} {#2}
    }
}
```

ate_execute_if_keytype_exist:nT
ate_execute_if_keytype_exist:VF

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:Npn \xtemplate_execute_if_keytype_exist:nT #1#2 {
  \cs_if_exist:cTF { xtemplate_store_value_ #1 :n }
    {#2}
    { \msg_kernel_error:nnx { xtemplate } { unknown-keytype } {#1} }
}
\cs_generate_variant:Nn \xtemplate_execute_if_keytype_exist:nT { V }
```

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

```
\cs_new:Npn \xtemplate_execute_if_type_exist:nT #1#2 {
  \prop_if_in:NnTF \g_xtemplate_object_type_prop {#1}
    {#2}
    { \msg_kernel_error:nnx { xtemplate } { unknown-object-type } {#1} }
}
```

`plate_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:Npn \xtemplate_if_keys_exist:nnT #1#2#3 {
  \cs_if_exist:cTF
    { g_ \c_xtemplate_keytypes_root_tl #1 / #2 _prop }
    {#3}
    {
      \msg_kernel_error:nnxx { xtemplate } { unknown-template }
        {#1} {#2}
    }
}
```

`\xtemplate_if_key_value:nT` Tests for the first token in a string being `\KeyValue`, where `\EvaluateNow` is not important.
`\xtemplate_if_key_value:VT`

```
\prg_set_conditional:Nnn \xtemplate_if_key_value:n { T } {
  \str_if_eq:noTF { \KeyValue } { \tl_head:w #1 \q_stop } {
    \prg_return_true:
  }{
    \prg_return_false:
  }
}
\cs_generate_variant:Nn \xtemplate_if_key_value:nT { V }
```

`\xtemplate_if_eval_now:nTF` Tests for the first token in a string being `\EvaluateNow`.

```
\prg_new_conditional:Nnn \xtemplate_if_eval_now:n { TF } {
  \str_if_eq:noTF { \EvaluateNow } { \tl_head:w #1 \q_stop } {
    \prg_return_true:
  }{
    \prg_return_false:
  }
}
```

emplate_if_instance_exist:nnn*TF* Testing for an instance is collection dependent.

```
\prg_new_conditional:Nnn \xtemplate_if_instance_exist:nnn { T, F, TF } {
  \cs_if_exist:cTF { \c_xtemplate_instances_root_tl #1 / #2 / #3 :w } {
    \prg_return_true:
  }{
    \prg_return_false:
  }
}
```

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

```
\prg_new_conditional:Nnn \xtemplate_if_use_template:n { TF } {
  \str_if_eq:noTF { \UseTemplate } { \tl_head:w #1 \q_stop } {
    \prg_return_true:
  }{
    \prg_return_false:
  }
}
```

9.1.1 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:Npn \xtemplate_store_defaults:n #1 {
  \cs_if_free:cT { g_ \c_xtemplate_defaults_root_tl #1 _prop } {
    \prop_new:c { g_ \c_xtemplate_defaults_root_tl #1 _prop }
  }
  \prop_gset_eq:cN { g_ \c_xtemplate_defaults_root_tl #1 _prop }
    \l_xtemplate_values_prop
}
\cs_new:Npn \xtemplate_store_keytypes:n #1 {
  \cs_if_free:cTF { g_ \c_xtemplate_keytypes_root_tl #1 _prop } {
    \msg_kernel_info:nnx { xtemplate } { define-template-interface }
      {#1}
    \prop_new:c { g_ \c_xtemplate_keytypes_root_tl #1 _prop }
  }
  {
    \msg_kernel_warning:nnx { xtemplate }
      { redefine-template-interface } {#1}
  }
  \prop_gset_eq:cN { g_ \c_xtemplate_keytypes_root_tl #1 _prop }
    \l_xtemplate_keytypes_prop
}
\cs_new:Npn \xtemplate_store_values:n #1 {
  \cs_if_free:cT { l_ \c_xtemplate_values_root_tl #1 _prop } {
```

```
    \prop_new:c { l_ \c_xtemplate_values_root_tl #1 _prop }
  }
  \prop_set_eq:cN { l_ \c_xtemplate_values_root_tl #1 _prop }
    \l_xtemplate_values_prop
}
\cs_new:Npn \xtemplate_store_restrictions:n #1 {
  \cs_if_free:cT { g_ \c_xtemplate_restrict_root_tl #1 _clist } {
    \clist_new:c { g_ \c_xtemplate_restrict_root_tl #1 _clist }
  }
  \clist_gset_eq:cN { g_ \c_xtemplate_restrict_root_tl #1 _clist }
    \l_xtemplate_restrict_clist
}
\cs_new:Npn \xtemplate_store_vars:n #1 {
  \cs_if_free:cTF { g_ \c_xtemplate_vars_root_tl #1 _prop } {
    \msg_kernel_info:nnx { xtemplate } { define-template-code } {#1}
    \prop_new:c { g_ \c_xtemplate_vars_root_tl #1 _prop }
  }
  {
    \msg_kernel_warning:nnx { xtemplate } { redefine-template-code }
      {#1}
  }
  \prop_gset_eq:cN { g_ \c_xtemplate_vars_root_tl #1 _prop }
    \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:Npn \xtemplate_recover_defaults:n #1 {
  \prop_set_eq:Nc \l_xtemplate_values_prop
    { g_ \c_xtemplate_defaults_root_tl #1 _prop }
}
\cs_new:Npn \xtemplate_recover_keytypes:n #1 {
  \prop_set_eq:Nc \l_xtemplate_keytypes_prop
    { g_ \c_xtemplate_keytypes_root_tl #1 _prop }
}
\cs_new:Npn \xtemplate_recover_restrictions:n #1 {
% FMi why is this called before being defined????
  \cs_if_free:cT { g_ \c_xtemplate_restrict_root_tl #1 _clist } {
    \clist_new:c { g_ \c_xtemplate_restrict_root_tl #1 _clist }
  }
  \clist_set_eq:Nc \l_xtemplate_restrict_clist
    { g_ \c_xtemplate_restrict_root_tl #1 _clist }
}
\cs_new:Npn \xtemplate_recover_values:n #1 {
  \prop_set_eq:Nc \l_xtemplate_values_prop
    { l_ \c_xtemplate_values_root_tl #1 _prop }
}
\cs_new:Npn \xtemplate_recover_vars:n #1 {
  \prop_set_eq:Nc \l_xtemplate_vars_prop
```

```
    { g_ \c_xtemplate_vars_root_tl #1 _prop }
}
```

9.1.2 Creating new object types

`template_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:Npn \xtemplate_declare_object_type:nn #1#2 {
  \int_set:Nn \l_xtemplate_tmp_int {#2}
  \bool_if:nTF {
    \int_compare_p:n { #2 > \c_nine } ||
    \int_compare_p:n { #2 < \c_zero }
  } {
    \msg_kernel_error:nnxx { xtemplate } { bad-number-of-arguments }
      {#1} { \exp_not:V \l_xtemplate_tmp_int }
  }{
    \prop_if_in:NnTF \g_xtemplate_object_type_prop {#1}
      {
        \msg_kernel_warning:nnxx { xtemplate } { redefine-object-type }
          {#1} {#2}
      }
      {
        \msg_kernel_info:nnxx { xtemplate } { define-object-type }
          {#1} {#2}
      }
    \prop_gput:NnV \g_xtemplate_object_type_prop {#1}
      \l_xtemplate_tmp_int
  }
}
```

9.2 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.

`late_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 l3keyval module to actually parse the keys. Finally, the code hands of to the storage routine to save the parsed information properly.

```
\cs_new: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_seq
       \KV_process_space_removal_sanitize:NNn
         \xtemplate_parse_keys_elt:n \xtemplate_parse_keys_elt:nn {#4}
       \prop_put:NVV \l_xtemplate_keytypes_prop \c_xtemplate_key_order_tl
 \l_xtemplate_key_seq
       \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:Npn \xtemplate_parse_keys_elt:n #1 {
  \xtemplate_split_keytype:n {#1}
  \bool_if:NF \l_xtemplate_error_bool
    {
      \xtemplate_execute_if_keytype_exist:VT \l_xtemplate_keytype_tl
        {
          \clist_map_function:nN { choice , function , instance }
            \xtemplate_parse_keys_elt_aux:n
          \bool_if:NF \l_xtemplate_error_bool
            {
              \seq_if_in:NVTF \l_xtemplate_key_seq
                \l_xtemplate_key_name_tl
                {
                  \msg_kernel_error:nnx { xtemplate }
                    { duplicate-key-interface }
                    { \l_xtemplate_key_name_tl }
                }
                { \xtemplate_parse_keys_elt_aux: }
            }
        }
    }
}
\cs_new_nopar:Npn \xtemplate_parse_keys_elt_aux:n #1 {
  \str_if_eq:VnT \l_xtemplate_keytype_tl {#1} {
    \tl_if_empty:NT \l_xtemplate_keytype_arg_tl {
      \msg_kernel_error:nnx { xtemplate }
```

```
        { keytype-requires-argument } {#1}
      \bool_set_true:N \l_xtemplate_error_bool
      \clist_map_break:
    }
  }
}
\cs_new_nopar:Npn \xtemplate_parse_keys_elt_aux: {
  \tl_set:Nx \l_xtemplate_tmp_tl {
    \l_xtemplate_keytype_tl
    \l_xtemplate_keytype_arg_tl
  }
  \prop_put:NVV \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
    \l_xtemplate_tmp_tl
  \seq_put_right:NV \l_xtemplate_key_seq \l_xtemplate_key_name_tl
  \str_if_eq:VnT \l_xtemplate_keytype_tl { choice } {
    \clist_if_in:NnT \l_xtemplate_keytype_arg_tl { unknown } {
      \msg_kernel_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. Choices have special handling as the code is not yet available to actually do the storing!

```
\cs_new:Npn \xtemplate_parse_keys_elt:nn #1#2 {
  \xtemplate_parse_keys_elt:n {#1}
  \str_if_eq:VnTF \l_xtemplate_keytype_tl { choice } {
    \xtemplate_store_value_choice_name:n {#2}
  }{
    \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_make_other:N \@
\tl_to_lowercase:n {
  \group_end:
  \cs_new: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_in:Nn \l_xtemplate_tmp_tl { ~ }
```

```
    \tl_replace_all_in:Nnn \l_xtemplate_tmp_tl { : } { @ }
    \tl_if_in:VnTF \l_xtemplate_tmp_tl { @ } {
      \exp_after:wN \xtemplate_split_keytype_aux:w \l_xtemplate_tmp_tl
        \q_stop
    }{
      \bool_set_true:N \l_xtemplate_error_bool
      \msg_kernel_error:nnx { xtemplate } { no-keytype } {#1}
    }
  }
  \cs_new:Npn \xtemplate_split_keytype_aux:w #1 @ #2 \q_stop {
    \tl_if_empty:nT {#1} {
      \msg_kernel_error:nnx { xtemplate } { empty-key-name } { @ #2 }
    }
    \tl_set:Nx \l_xtemplate_key_name_tl { \tl_to_str:n {#1} }
    \xtemplate_split_keytype_arg:n {#2}
  }
}
```

\xtemplate_split_keytype_arg:n
\xtemplate_split_keytype_arg:V
\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: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_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}
            \clist_map_break:
        }
      }
      \xtemplate_split_keytype_arg_aux:w #1 \q_stop
    }
  }
  \clist_map_function:NN \c_xtemplate_keytypes_arg_clist
    \xtemplate_split_keytype_arg_aux:n
}
\cs_generate_variant:Nn \xtemplate_split_keytype_arg:n { V }
\cs_new_nopar:Npn \xtemplate_split_keytype_arg_aux:n #1 { }
\cs_new_nopar:Npn \xtemplate_split_keytype_arg_aux:w #1 \q_stop { }
```

9.2.1 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:Npn \xtemplate_store_value_boolean:n #1 {
  \xtemplate_if_eval_now:nTF {#1} {
    \bool_if:cTF { c_ #1 _bool } {
      \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl
        { true }
    }{
      \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl
        { false }
    }
  }{
    \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
  }
}
```

`\xtemplate_store_value_choice:n` `\xtemplate_store_value_choice:V` `mplate_store_value_choice_aux:n` `mplate_store_value_choice_aux:V` `plate_store_value_choice_name:n` Choices are a bit odd, as they have to be handled in two parts. When an interface is being created, the default is stored with a hidden name (using spaces and letter category codes). When a choice is actually being used, there is a check for the choice itself, then code to handle an unknown before issuing an error.

```
\cs_new:Npn \xtemplate_store_value_choice:n #1 {
  \tl_set:Nx \l_xtemplate_tmp_tl
    { \l_xtemplate_key_name_tl \c_space_tl #1 }
  \prop_if_in:NVTF \l_xtemplate_vars_prop \l_xtemplate_tmp_tl {
    \xtemplate_store_value_choice_aux:V \l_xtemplate_tmp_tl
  }{
    \tl_set:Nx \l_xtemplate_tmp_tl
      { \l_xtemplate_key_name_tl \c_space_tl unknown }
    \prop_if_in:NVTF \l_xtemplate_vars_prop \l_xtemplate_tmp_tl {
      \xtemplate_store_value_choice_aux:V \l_xtemplate_tmp_tl
    }{
      \prop_get:NVN \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
        \l_xtemplate_tmp_tl
```

```
      \xtemplate_split_keytype_arg:V \l_xtemplate_tmp_tl
      \msg_kernel_error:nnxxx { xtemplate } { unknown-choice }
        {#1}
        { \l_xtemplate_key_name_tl }
        { \l_xtemplate_keytype_arg_tl }
      \prop_gdel:NV \l_xtemplate_values_prop \l_xtemplate_key_name_tl
    }
  }
}
\cs_generate_variant:Nn \xtemplate_store_value_choice:n { V }
\cs_new:Npn \xtemplate_store_value_choice_aux:n #1 {
  \prop_get:NnN \l_xtemplate_vars_prop {#1} \l_xtemplate_tmp_tl
  \prop_put:NVV \l_xtemplate_values_prop \l_xtemplate_key_name_tl
    \l_xtemplate_tmp_tl
}
\cs_generate_variant:Nn \xtemplate_store_value_choice_aux:n { V }
\cs_new:Npn \xtemplate_store_value_choice_name:n #1 {
  \tl_set:Nx \l_xtemplate_tmp_tl
    { \l_xtemplate_key_name_tl \c_space_tl default }
  \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_tmp_tl {#1}
}
```

`\xtemplate_store_value_code:n`
`emplate_store_value_commalist:n`
`template_store_value_function:n`
`template_store_value_instance:n`
`emplate_store_value_tokenlist:n`

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

```
\cs_new:Npn \xtemplate_store_value_code:n #1 {
  \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
}
\cs_new:Npn \xtemplate_store_value_commalist:n #1 {
  \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
}
\cs_new:Npn \xtemplate_store_value_function:n #1 {
  \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
}
\cs_new:Npn \xtemplate_store_value_instance:n #1 {
  \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
}
\cs_new:Npn \xtemplate_store_value_tokenlist:n #1 {
  \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
}
```

`xtemplate_store_value_integer:n`
`\xtemplate_store_value_length: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: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_tl
```

```
      \l_xtemplate_tmp_int
  }{
    \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
  }
}
\cs_new: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:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
  }
}
\cs_new: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:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
  }
}
```

9.2.2 Implementation part of template declaration

`ate_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: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 :w } \cs_gset:Npn
              {#3} {#5}
          }
       }
    }
}
```

`te_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_set: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
  \KV_process_space_removal_sanitize:NNn
    \xtemplate_parse_vars_elt:n \xtemplate_parse_vars_elt:nn {#3}
```

In certain cases (when choices are involved) parsing the key results in changes to the default values. Therefore we have to save those back.

```
  \xtemplate_store_defaults:n { #1 / #2 }
  \xtemplate_store_vars:n { #1 / #2 }
  \clist_clear:N \l_xtemplate_restrict_clist
  \xtemplate_store_restrictions:n { #1 / #2 }
  \prop_del:NV \l_xtemplate_keytypes_prop \c_xtemplate_key_order_tl
  \prop_if_empty:NF \l_xtemplate_keytypes_prop {
    \prop_map_inline:Nn \l_xtemplate_keytypes_prop {
      \msg_kernel_error:nnxxx { xtemplate } { key-not-implemented }
        {##1} {#2} {#1}
    }
  }
}
```

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

```
\cs_new:Npn \xtemplate_parse_vars_elt:n #1 {
  \msg_kernel_error:nnx { xtemplate } { key-no-variable } {#1}
}
```

`\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:Npn \xtemplate_parse_vars_elt:nn #1#2 {
  \tl_set:Nx \l_xtemplate_key_name_tl { \tl_to_str:n {#1} }
  \tl_replace_all_in:Nnn \l_xtemplate_key_name_tl { ~ } { }
  \prop_if_in:NVTF \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl {
    \prop_get:NVN \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
      \l_xtemplate_keytype_tl
    \xtemplate_split_keytype_arg:V \l_xtemplate_keytype_tl
    \xtemplate_parse_vars_elt_aux:n {#2}
    \prop_del:NV \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
  }{
    \msg_kernel_error:nnx { xtemplate } { unknown-key } {#1}
  }
}
```

\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:Npn \xtemplate_parse_vars_elt_aux:n #1 {
  \str_if_eq:VnTF \l_xtemplate_keytype_tl { choice }
    { \xtemplate_implement_choices:n {#1} }
    {
      \str_if_eq:VnTF \l_xtemplate_keytype_tl { code }
        {
          \prop_put:NVn \l_xtemplate_vars_prop
            \l_xtemplate_key_name_tl {#1}
        }
        {
          \str_if_eq:noTF {#1} { \tl_head:w #1 \q_stop }
            {
              \cs_if_exist:NTF #1
                {
                  \prop_put:NVn \l_xtemplate_vars_prop
                    \l_xtemplate_key_name_tl {#1}
                }
                {
                  \msg_kernel_error:nnx { xtemplate }
                    { undeclared-variable }
                    { \exp_not:N #1 }
                }
            }
            {
              \tl_if_in:nnTF {#1} { global }
                { \xtemplate_parse_vars_elt_aux:w #1 \q_stop }
                {
                  \msg_kernel_error:nnx { xtemplate } { bad-variable }
                    { \exp_not:n {#1} }
                }
            }
        }
    }
}
\cs_new:Npn \xtemplate_parse_vars_elt_aux:w #1 global #2 \q_stop {
  \tl_if_empty:nTF {#1}
    {
      \str_if_eq:noTF {#2} { \tl_head:w #2 \q_stop }
        {
          \cs_if_exist:NTF #2
            {
              \prop_put:NVn \l_xtemplate_vars_prop
                \l_xtemplate_key_name_tl { #1 global #2 }
            }
            {
              \msg_kernel_error:nnx { xtemplate }
```

```
                { undeclared-variable }
                { \exp_not:N #2 }
            }
        }
        {
          \msg_kernel_error:nnx { xtemplate } { bad-variable }
            { \exp_not:n { #1 global #2 } }
        }
    }
    {
        \msg_kernel_error:nnx { xtemplate } { bad-variable }
          { \exp_not:n { #1 global #2 } }
    }
}
```

`\xtemplate_implement_choices:n` Implementing choices requires a second key–value loop. So after a little set-up, the standard parser is called. There is then a check for a default choice being set: at this stage the name of the choice is replaced by the code to implement it.

```
\cs_new:Npn \xtemplate_implement_choices:n #1 {
  \clist_set_eq:NN \l_xtemplate_tmp_clist \l_xtemplate_keytype_arg_tl
  \prop_put:NVn \l_xtemplate_vars_prop \l_xtemplate_key_name_tl { }
  \KV_process_space_removal_sanitize:NNn
    \xtemplate_implement_choice_elt:n \xtemplate_implement_choice_elt:nn
    {#1}
  \tl_set:Nx \l_xtemplate_tmp_tl
    { \l_xtemplate_key_name_tl \c_space_tl default }
  \prop_if_in:NVT \l_xtemplate_values_prop \l_xtemplate_tmp_tl {
    \prop_get:NVN \l_xtemplate_values_prop \l_xtemplate_tmp_tl
      \l_xtemplate_tmp_tl
    \xtemplate_store_value_choice:V \l_xtemplate_tmp_tl
  }
  \clist_if_empty:NF \l_xtemplate_tmp_clist {
    \clist_map_inline:Nn \l_xtemplate_tmp_clist
      {
        \msg_kernel_error:nnx { xtemplate } { choice-not-implemented }
          {##1}
      }
  }
}
```

`template_implement_choice_elt:n`
`emplate_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:Npn \xtemplate_implement_choice_elt:n #1 {
  \clist_if_empty:NTF \l_xtemplate_tmp_clist {
    \str_if_eq:nnF {#1} { unknown } {
```

```
      \prop_get:NVN \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
        \l_xtemplate_tmp_tl
      \xtemplate_split_keytype_arg:V \l_xtemplate_tmp_tl
      \msg_kernel_error:nnxxx { xtemplate } { unknown-choice }
        {#1}
        { \l_xtemplate_key_name_tl }
        { \l_xtemplate_keytype_arg_tl }
    }
  }{
    \clist_if_in:NnTF \l_xtemplate_tmp_clist {#1} {
      \clist_remove_element:Nn \l_xtemplate_tmp_clist {#1}
    }{
      \prop_get:NVN \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
        \l_xtemplate_tmp_tl
      \xtemplate_split_keytype_arg:V \l_xtemplate_tmp_tl
      \msg_kernel_error:nnxxx { xtemplate } { unknown-choice }
        {#1}
        { \l_xtemplate_key_name_tl }
        { \l_xtemplate_keytype_arg_tl }
    }
  }
}
\cs_new: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:NVn \l_xtemplate_vars_prop \l_xtemplate_tmp_tl {#2}
}
```

9.2.3 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.

`emplate_declare_restricted:nnnn` Creating a restricted template means copying the old template to the new one first.

```
\cs_new: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}
    }
}
```

`\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:Npn \xtemplate_edit_defaults:nnn {
  \bool_set_false:N \l_xtemplate_restrict_bool
  \xtemplate_edit_defaults_aux:nnn
}
\cs_new: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 }
    }
}
```

`\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:Npn \xtemplate_parse_values:nn #1#2 {
  \xtemplate_recover_keytypes:n {#1}
  \clist_clear:N \l_xtemplate_restrict_clist
  \KV_process_space_removal_sanitize:NNn
    \xtemplate_parse_values_elt:n \xtemplate_parse_values_elt:nn {#2}
}
```

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

```
\cs_new:Npn \xtemplate_parse_values_elt:n #1 {
  \bool_set_true:N \l_xtemplate_error_bool
  \msg_kernel_error:nnx { xtemplate } { key-no-value } {#1}
}
```

`\xtemplate_parse_values_elt:nn`
`emplate_parse_values_elt_aux:nn`
`emplate_parse_values_elt_aux:Vn` 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:Npn \xtemplate_parse_values_elt:nn #1#2 {
  \tl_set:Nx \l_xtemplate_key_name_tl { \tl_to_str:n {#1} }
  \tl_replace_all_in:Nnn \l_xtemplate_key_name_tl { ~ } { }
  \prop_if_in:NVTF \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl {
    \bool_if:NTF \l_xtemplate_restrict_bool {
      \clist_if_in:NVF \l_xtemplate_restrict_clist
        \l_xtemplate_key_name_tl {
        \xtemplate_parse_values_elt_aux:Vn \l_xtemplate_key_name_tl {#2}
      }
    }{
      \xtemplate_parse_values_elt_aux:Vn \l_xtemplate_key_name_tl {#2}
```

```
    }
  }{
    \msg_kernel_error:nnx { xtemplate } { unknown-key }
      { \l_xtemplate_key_name_tl }
  }
}
\cs_new:Npn \xtemplate_parse_values_elt_aux:nn #1#2 {
  \clist_put_right:Nn \l_xtemplate_restrict_clist  {#1}
  \prop_get:NnN \l_xtemplate_keytypes_prop {#1} \l_xtemplate_tmp_tl
  \xtemplate_split_keytype_arg:V \l_xtemplate_tmp_tl
  \use:c { xtemplate_store_value_ \l_xtemplate_keytype_tl :n } {#2}
}
\cs_generate_variant:Nn \xtemplate_parse_values_elt_aux:nn { Vn }
```

`\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: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 :w }
    { \c_xtemplate_code_root_tl #2 :w }
}
```

9.2.4 Creating instances of templates

`template_declare_instance:nnnnn`
`late_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: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: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:cpx { \c_xtemplate_instances_root_tl #1 / #3 / #4 :w } {
      \exp_not:N \xtemplate_assignments_push:n {
        \exp_not:V \l_xtemplate_assignments_tl
      }
      \exp_not:c { \c_xtemplate_code_root_tl #1 / #2 :w }
    }
    \xtemplate_if_instance_exist:nnnF {#1} { } {#4} {
      \cs_set_eq:cc
        { \c_xtemplate_instances_root_tl #1 /    / #4 :w }
        { \c_xtemplate_instances_root_tl #1 / #3 / #4 :w }
    }
  }
}
```

`\xtemplate_edit_instance:nnnn`
`\xtemplate_edit_instance_aux:nnnnn`
`\xtemplate_edit_instance_aux:nVnnn`

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: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:nVnnn {#1} \l_xtemplate_tmp_tl
        {#2} {#3}
    }
    {
      \msg_kernel_error:nnxx { xtemplate } { unknown-instance }
        {#1} {#3}
    }
}
\cs_new: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 { nV }
```

`emplate_convert_to_assignments:`
`te_convert_to_assignments_aux:n`
`e_convert_to_assignments_aux:nn`
`e_convert_to_assignments_aux:nV`

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

```
\cs_new_nopar:Npn \xtemplate_convert_to_assignments: {
  \tl_clear:N \l_xtemplate_assignments_tl
  \prop_get:NVN \l_xtemplate_keytypes_prop \c_xtemplate_key_order_tl
    \l_xtemplate_key_seq
```

```
  \seq_map_function:NN \l_xtemplate_key_seq
    \xtemplate_convert_to_assignments_aux:n
}
\cs_new: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:nV {#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:Npn \xtemplate_convert_to_assignments_aux:nn #1#2 {
  \prop_if_in:NnT \l_xtemplate_values_prop {#1} {
    \prop_if_in:NnTF \l_xtemplate_vars_prop {#1} {
      \prop_get:NnN \l_xtemplate_values_prop {#1} \l_xtemplate_value_tl
      \prop_get:NnN \l_xtemplate_vars_prop {#1} \l_xtemplate_var_tl
      \xtemplate_split_keytype_arg:n {#2}
      \str_if_eq:VnF \l_xtemplate_keytype_tl { choice } {
        \str_if_eq:VnF \l_xtemplate_keytype_tl { code } {
          \xtemplate_find_global:
        }
      }
      \use:c { xtemplate_assign_ \l_xtemplate_keytype_tl : }
    }{
      \msg_kernel_error:nnx { xtemplate } { unknown-attribute } {#1}
    }
  }
}
\cs_generate_variant:Nn \xtemplate_convert_to_assignments_aux:nn { nV }
```

`\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_nopar:Npn \xtemplate_find_global: {
  \bool_set_false:N \l_xtemplate_global_bool
  \tl_if_in:VnT \l_xtemplate_var_tl { global } {
    \exp_after:wN \xtemplate_find_global_aux:w \l_xtemplate_var_tl \q_stop
  }
}
\cs_new: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
}
```

9.3 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:Npn \xtemplate_use_template:nnn #1#2#3 {
  \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 :w  }
}
```

9.3.1 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_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_nopar:Npn \xtemplate_assign_boolean_aux:n #1 {
  \xtemplate_if_key_value:VT \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:V \l_xtemplate_var_tl
  }
}
```

\xtemplate_assign_choice:

Assigning a choice is actually trivial: the code needed will be in `\l_xtemplate_value_tl`, and is simply copied to the correct place.

```
\cs_new_nopar:Npn \xtemplate_assign_choice: {
  \tl_put_left:NV \l_xtemplate_assignments_tl \l_xtemplate_value_tl
}
```

\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_nopar:Npn \xtemplate_assign_code: {
  \tl_put_left:Nx \l_xtemplate_assignments_tl {
    \exp_not:N \cs_set:Npn \exp_not:N \xtemplate_assign_code:n
      \exp_not:n {##1}
        { \exp_not:V \l_xtemplate_var_tl }
    \exp_not:N \xtemplate_assign_code:n
      { \exp_not:V \l_xtemplate_value_tl }
  }
}
\cs_new:Npn \xtemplate_assign_code:n #1 { }
```

`\xtemplate_assign_function:`
`\xtemplate_assign_function_aux:N`

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

```
\cs_new_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_nopar:Npn \xtemplate_assign_function_aux:N #1 {
  \tl_put_left:Nx \l_xtemplate_assignments_tl {
    \exp_not:N \cs_generate_from_arg_count:NNnn
      \exp_not:V \l_xtemplate_var_tl
      \exp_not:N #1
      { \exp_not:V \l_xtemplate_keytype_arg_tl }
      { \exp_not:V \l_xtemplate_value_tl }
  }
}
```

`\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_nopar:Npn \xtemplate_assign_instance: {
  \bool_if:NTF \l_xtemplate_global_bool {
    \xtemplate_assign_instance_aux:N \cs_gset:Npn
  }{
    \xtemplate_assign_instance_aux:N \cs_set:Npn
  }
}
\cs_new_nopar:Npn \xtemplate_assign_instance_aux:N #1 {
  \tl_put_left:Nx \l_xtemplate_assignments_tl {
    \exp_not:N #1 \exp_not:V \l_xtemplate_var_tl {
      \exp_not:N \xtemplate_use_instance:nn
        { \exp_not:V \l_xtemplate_keytype_arg_tl }
        { \exp_not:V \l_xtemplate_value_tl }
    }
  }
}
```

\xtemplate_assign_integer:
\xtemplate_assign_length:
\xtemplate_assign_skip:

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

```
\cs_new_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_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_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
  }
}
```

\xtemplate_assign_tokenlist:
\xtemplate_assign_commalist:
\xtemplate_assign_tokenlist_aux:N

Storing lists of tokens is easy: no complex calculations and no need to worry about numbers of arguments. The comma list version takes advantage of the low-level implementation of the variable type to keep down code duplication.

```
\cs_new_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_eq:NN \xtemplate_assign_commalist:
  \xtemplate_assign_tokenlist:
\cs_new_nopar:Npn \xtemplate_assign_tokenlist_aux:N #1 {
  \tl_put_left:Nx \l_xtemplate_assignments_tl {
    \exp_not:N #1 \exp_not:V \l_xtemplate_var_tl
      { \exp_not:V \l_xtemplate_value_tl }
  }
}
```

\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_nopar:Npn \xtemplate_assign_variable:N #1 {
```

```
  \xtemplate_if_key_value:VT \l_xtemplate_value_tl {
    \xtemplate_key_to_value:
  }
  \tl_put_left:Nx \l_xtemplate_assignments_tl {
    \exp_not:N #1 \exp_not:V \l_xtemplate_var_tl
       { \exp_not:V \l_xtemplate_value_tl }
  }
}
```

`\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_nopar:Npn \xtemplate_key_to_value: {
  \exp_after:wN \xtemplate_key_to_value_aux:w \l_xtemplate_value_tl
}
\cs_new:Npn \xtemplate_key_to_value_aux:w \KeyValue #1 {
  \tl_set:Nx \l_xtemplate_tmp_tl { \tl_to_str:n {#1} }
  \tl_replace_all_in:Nnn \l_xtemplate_key_name_tl { ~ } { }
  \prop_if_in:NVTF \l_xtemplate_vars_prop \l_xtemplate_tmp_tl {
    \prop_get:NVN \l_xtemplate_vars_prop \l_xtemplate_tmp_tl
       \l_xtemplate_value_tl
  }{
    \msg_kernel_error:nnx { xtemplate } { unknown-attribute }
      { \l_xtemplate_tmp_tl }
  }
}
```

9.3.2 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: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: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_kernel_error:nnxx { xtemplate } { type-mismatch } {#1} {#3}
  }
}
\cs_new: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
      :w
    }
  }{
    \xtemplate_if_instance_exist:nnnTF {#1} { } {#2} {
      \use:c { \c_xtemplate_instances_root_tl #1 / / #2 :w }
    }{
      \msg_kernel_error:nnxx { xtemplate } { unknown-instance }
        {#1} {#2}
    }
  }
}
```

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

```
\cs_new:Npn \xtemplate_use_collection:nn #1#2 {
  \prop_put:Nnn \l_xtemplate_collections_prop {#1} {#2}
}
```

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

```
\cs_new:Npn \xtemplate_get_collection:n #1 {
  \prop_if_in:NnTF \l_xtemplate_collections_prop {#1} {
    \prop_get:NnN \l_xtemplate_collections_prop {#1}
      \l_xtemplate_collection_tl
  }{
    \tl_clear:N \l_xtemplate_collection_tl
  }
}
```

9.3.3 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: {
  \tl_use:N \l_xtemplate_assignments_tl
}
```

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

```
\cs_new:Npn \xtemplate_assignments_push:n #1 {
  \tl_set:Nn \l_xtemplate_assignments_tl {#1}
}
```

9.3.4 Showing templates and instances

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

```
\cs_new:Npn \xtemplate_show_code:nn #1#2 {
  \xtemplate_execute_if_code_exist:nnT {#1} {#2}
    { \cs_show:c { \c_xtemplate_code_root_tl #1 / #2 :w } }
}
```

\xtemplate_show_defaults:nn
\xtemplate_show_keytypes:nn
\xtemplate_show_values:nnn
\xtemplate_show_vars:nn
Showing the internal data is a case of getting the appropriate property list back, then displaying the scratch variable.

```
\cs_new:Npn \xtemplate_show_defaults:nn #1#2 {
  \xtemplate_if_keys_exist:nnT {#1} {#2}
    {
      \xtemplate_recover_defaults:n { #1 / #2 }
      \prop_display:N \l_xtemplate_values_prop
    }
}
\cs_new:Npn \xtemplate_show_keytypes:nn #1#2 {
  \xtemplate_if_keys_exist:nnT {#1} {#2}
    {
      \xtemplate_recover_keytypes:n { #1 / #2 }
      \prop_display:N \l_xtemplate_keytypes_prop
    }
}
\cs_new:Npn \xtemplate_show_values:nnn #1#2#3 {
  \xtemplate_if_instance_exist:nnnTF {#1} {#2} {#3}
    {
      \xtemplate_recover_values:n { #1 / #2 / #3 }
      \prop_display:N \l_xtemplate_values_prop
    }
    {
      \msg_kernel_error:nnxx { xtemplate } { unknown-instance }
        {#1} {#2}
    }
}
\cs_new:Npn \xtemplate_show_vars:nn #1#2 {
  \xtemplate_execute_if_code_exist:nnT {#1} {#2}
    {
      \xtemplate_recover_vars:n { #1 / #2 }
      \prop_display:N \l_xtemplate_vars_prop
    }
}
```

9.3.5 Messages

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

```
\msg_kernel_new:nnnn { xtemplate } { argument-number-mismatch }
  { Object~type~'#1'~takes~'#2'~not~'#3'~argument(s). }
  {
    Objects~of~type~'#1'~require~'#2'~arguments.\\
    You~have~tried~to~make~a~template~for~'#1'\\
    with~'#3'~arguments,~which~is~not~possible:\\
    the~number~of~arguments~must~agree.%
  }
\msg_kernel_new:nnnn { xtemplate } { bad-number-of-arguments }
  {
    Impossible~number~of~arguments~declared~for \\
    object~type~'#1'.
  }
  {
    An~object~may~accept~between~0~and~9~arguments.\\
    You~asked~to~use~#2~arguments:~this~is~not~supported.
  }
\msg_kernel_new:nnnn { xtemplate } { bad-variable }
  { Incorrect~variable~description~\msg_line_context:. }
  {
    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_kernel_new:nnnn { xtemplate } { choice-not-implemented }
  { The~choice~'#1'~has~no~implementation~\msg_line_context:. }
  {
    Each~choice~listed~in~the~interface~for~a~template~must\\
    have~an~implementation.\\
    I~did~not~find~an~implementation~for~'#1'.
  }
\msg_kernel_new:nnnn { xtemplate } { choice-unknown-reserved }
  { The~choice~'unknown'~cannot~be~given~for~a~template~key.}
  {
    The~'unknown'~choice~is~automatically~checked~by~template~when \\
    a~choice~key~is~given~with~an~unknown~value.~It~should~not~be \\
    included~in~the~list~of~named~choices~for~a~key,~and~is~always \\
    given~last~in~the~implementation~of~choices.
  }
\msg_kernel_new:nnnn { xtemplate } { duplicate-key-interface }
  { Key~'#1'~appears~twice~in~interface~definition~\msg_line_context:. }
  {
    Each~key~can~only~have~one~interface~declared~in~a~template.\\
    I~found~two~interfaces~for~'#1':~probably~a~typing~error.
  }
\msg_kernel_new:nnnn { xtemplate } { empty-key-name }
```

```
  { No~key~name~found~in~'#1'~\msg_line_context:. }
  {
    A~template~key~name~and~type~is~given~in~the~form:\\
    \c_space_tl <name>~:~<type> \\
    Parsing~your~input~I~found~a~:'~but~nothing~before~it!
  }
\msg_kernel_new:nnnn { xtemplate } { key-no-value }
  { The~key~'#1'~has~no~value~\msg_line_context:.}
  {
    When~creating~an~instance~of~a~template\\
    every~key~listed~must~include~a~value:
    \c_space_tl \c_space_tl <key>~=~<value>
  }
\msg_kernel_new:nnnn { xtemplate } { key-no-variable }
  { The~key~'#1'~requires~implementation~details~\msg_line_context:. }
  {
    When~creating~template~code~using~\DeclareTemplateCode,\\
    each~key~name~must~have~an~associated~implementation.\\
    This~should~be~given~after~a~'='~sign:~I~did~not~find~one.
  }
\msg_kernel_new:nnnn { xtemplate } { key-not-implemented }
  { Key~'#1'~has~no~implementation~\msg_line_context:. }
  {
    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_kernel_new:nnnn { xtemplate } { keytype-requires-argument }
  { The~keytype~'#1'~requires~an~argument~\msg_line_context:. }
  {
    You~should~have~put:\\
    \c_space_tl \c_space_tl <key-name>~:~#1~
      \token_to_str:N {~<argument>~\token_to_str:N } \\
    but~I~did~not~find~an~<argument>.
  }
\msg_kernel_new:nnnn { xtemplate } { no-keytype }
  { The~key~'#1'~is missing~a~keytype~\msg_line_context:. }
  {%
    You~should~have~put:\\
    \c_space_tl \c_space_tl #1~:~<keytype>\\
    but~I~did~not~find~a~<keytype>.
  }
\msg_kernel_new:nnnn { xtemplate } { no-template-code }
  {
    The~template~'#2'~of~type~'#1'~is~unknown\\
    or~has~no~implementation.
  }
  {
    You~need~to~declare~a~template~with~\DeclareTemplateInterface \\
```

```
    and~its~code~with~\DeclareTemplateCode prior~to~using~it.
  }
\msg_kernel_new:nnnn { xtemplate } { 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_kernel_new:nnnn { xtemplate } { undeclared-variable }
  { The~variable~#1~has~not~been~declared. }
  {
    Before~declaring~an~instance,~all~of~the~required~variables\\
    used~in~template~keys~must~be~created.
  }
\msg_kernel_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
    \token_to_str:N {~#1~\token_to_str:N }\\
    but~there~is~no~key~called~'#1'.
  }
\msg_kernel_new:nnnn { xtemplate } { unknown-choice }
  { The~choice~'#1'~was~not~declared~for~key~'#2'~\msg_line_context:. }
  {
    The~key~'#2'~takes~a~fixed~number~of~choices:\\
    \clist_map_function:NN #3 \xtemplate_unknown_choice_aux:n
    and~this~list~does~not~include~'#1'.
  }
\cs_new:Npn \xtemplate_unknown_choice_aux:n #1 { ~~ #1 ;\\}
\msg_kernel_new:nnnn { xtemplate } { unknown-keytype }
  { The~keytype~'#1'~is~unknown~\msg_line_context:. }
  {
    Valid~keytypes~are:\\
    ~~boolean;\\
    ~~choice;\\
    ~~code;\\
    ~~commalist;\\
    ~~function;\\
    ~~instance;\\
    ~~integer;\\
    ~~length;\\
    ~~skip;\\
    ~~tokenlist.
  }
\msg_kernel_new:nnnn { xtemplate } { unknown-object-type }
  { The~object~type~'#1'~is~unknown~\msg_line_context:. }
  {
    An~object~type~needs~to~be~declared~with~
    \DeclareObjectType prior~to~using~it.
```

```
  }
\msg_kernel_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_kernel_new:nnnn { xtemplate } { unknown-key }
  { Unknown~template~key~'#1'~\msg_line_context:. }
  {
    The~key~'#1'~was~not~declared~in~the~interface\\
    for~the~current~template.\\
    Probably~you~have~misspelled~the~key~name.
  }
\msg_kernel_new:nnnn { xtemplate } { unknown-template }
  { The~template~'#2'~of~type~'#1'~is~unknown~\msg_line_context:. }
  {
    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_kernel_new:nnn { xtemplate } { define-template-code }
  {Defining~template~code~for~'#1'~\msg_line_context:.}
\msg_kernel_new:nnn { xtemplate } { define-template-interface }
  {Defining~template~interface~for~'#1'~\msg_line_context:.}
\msg_kernel_new:nnn { xtemplate } { define-object-type }
  {Defining~object~type~'#1'~with~#2~argument(s)~\msg_line_context:.}
\msg_kernel_new:nnn { xtemplate } { redefine-template-code }
  {Redefining~template~code~for~'#1'~\msg_line_context:.}
\msg_kernel_new:nnn { xtemplate } { redefine-template-interface }
  {Redefining~template~interface~for~'#1'~\msg_line_context:.}
\msg_kernel_new:nnn { xtemplate } { redefine-object-type }
  {Redefining~object~type~'#1'~with~#2~argument(s)~\msg_line_context:.}
```

9.3.6 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}
}
```

\ShowTemplateCode
\ShowTemplateDefaults
\ShowTemplateKeytypes
\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 \ShowTemplateKeytypes #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}
```

```
}
```

`\IfInstanceExist`*TF* 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}
}
```

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

They need to be expandable as they might get called in the context of setting some register value.

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

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

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

9.3.7 Recent additions to the code

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

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

⟨/initex | package⟩