

# The `xtemplate` package<sup>\*</sup>

The L<sup>A</sup>T<sub>E</sub>X3 Project<sup>†</sup>

2009/11/16

## 1 Introduction

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

1. Authoring of the text, with mark-up
2. Document layout design
3. Implementation (with T<sub>E</sub>X programming) of the design

We write the text as an author, and we see the visual output of the design after the document is generated; the T<sub>E</sub>X implementation in the middle is the glue between the two.

L<sup>A</sup>T<sub>E</sub>X’s greatest success has been to standardise a system of mark-up that balances the trade-off between ease of reading and ease of writing to suit almost all forms of technical writing. It’s other original strength was a good background in typographical design; while the standard L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> classes look somewhat dated now in terms of their visual design, their typography is generally sound. (Barring the occasional minor faults.)

However, L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> 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.

---

<sup>\*</sup>This file has version number 1746, last revised 2009/11/16.

<sup>†</sup>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  $\text{\LaTeX}$  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 key1> : <key type1> ,  
  <name of key2> : <key type2> = <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 key1⟩ = ⟨internal variable or code1⟩ ,
  ⟨name of key2⟩ = ⟨internal variable or code2⟩ ,
  ...
}{
  ⟨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.)<sup>1</sup> 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 key1⟩ =           ⟨internal variable1⟩ ,
⟨name of key2⟩ = global ⟨internal variable2⟩ ,
```

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.

---

<sup>1</sup>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
<code>boolean</code>	<code>true</code> or <code>false</code>
<code>choice</code> $\{\langle\textit{choices}\rangle\}$	A list of pre-defined choices
<code>code</code>	Generalised key type; use <code>#1</code> as the input to the key
<code>commalist</code>	A comma-separated list of arbitrary items
<code>function</code> $N$	A function definition with $N$ arguments ( $N$ from 0 to 9)
<code>instance</code> $\{\langle\textit{name}\rangle\}$	An instance of type $\langle\textit{name}\rangle$
<code>integer</code>	An integer expression (e.g., $(1 + 5)/2$ )
<code>length</code>	A dimension expression (e.g., <code>3pt + 2cm</code> )
<code>skip</code>	A dimension expression with glue (e.g., <code>3pt plus 2pt minus 1pt</code> )
<code>tokenlist</code>	A ‘token list’ input; any text or commands

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

Key Type	Description of binding
<code>boolean</code>	$\star$ Boolean variable; e.g., <code>\l_tmpa_bool</code>
<code>choice</code>	$\{ \langle\textit{choice}_1\rangle = \langle\textit{code}_1\rangle , \langle\textit{choice}_2\rangle = \langle\textit{code}_2\rangle , \dots \}$
<code>code</code>	$\langle\textit{code}\rangle$ ; use <code>#1</code> as the input to the key
<code>commalist</code>	$\star$ Comma-list variable; e.g., <code>\l_tmpa_clist</code>
<code>function</code>	$\star$ Function w/ $N$ arguments; e.g., <code>\use_i:nn</code>
<code>instance</code>	$\star$ An instance variable; e.g., <code>\g_foo_instance</code>
<code>integer</code>	$\star$ Integer variable; e.g., <code>\l_tmpa_int</code>
<code>length</code>	$\star$ Dimension variable; e.g., <code>\l_tmpa_dim</code>
<code>skip</code>	$\star$ Skip variable; e.g., <code>\l_tmpa_skip</code>
<code>tokenlist</code>	$\star$ Token list variable; e.g., <code>\l_tmpa_tl</code>

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 the template.

### 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`  $\{\langle object\ type\rangle\}$   $\{\langle instance\rangle\}$   $\{\langle template\rangle\}$   $\{\langle parameters\rangle\}$

The name of the instance being declared is  $\langle instance\rangle$ , with  $\langle parameters\rangle$  the keyval input to set some or all of the  $\langle template\rangle$  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`  $\{\langle object\ type\rangle\}$   $\{\langle instance\rangle\}$   $\langle arguments\rangle$

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`  $\{ \langle object\ type \rangle \}$   $\{ \langle template \rangle \}$   $\{ \langle settings \rangle \}$   $\langle arguments \rangle$

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 a 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?

`\IfInstanceExistTF`

```

\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.

<code>\EditInstance</code>
<code>\EditCollectionInstance</code>

```
\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

<code>\EvaluateNow</code>
---------------------------

`\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.

<code>\KeyValue</code>
------------------------

`\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 L<sup>A</sup>T<sub>E</sub>X 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.

<code>\DeclareRestrictedTemplate</code>
---

`\DeclareRestrictedTemplate`  $\{\langle object\ type\rangle\}$   $\{\langle parent\rangle\}$   $\{\langle new\ template\rangle\}$   $\{\langle frozen\ parameters\rangle\}$

Defines  $\langle new\ template\rangle$  based on template  $\langle parent\rangle$  (of certain  $\langle object\ type\rangle$ ) with certain keys set and frozen as defined in  $\langle keyvals\rangle$ .

## 6 Getting information about templates and instances

<code>\ShowTemplateCode</code> <code>\ShowTemplateDefaults</code> <code>\ShowTemplateKeytypes</code> <code>\ShowTemplateVariables</code>
---

`\ShowTemplateCode`  $\{\langle object\ type\rangle\}$   $\{\langle template\rangle\}$

These functions pause the typesetting and display in the console the various pieces of information for a template.

<code>\ShowInstanceValues</code> <code>\ShowCollectionInstanceValues</code>
--

`\ShowInstanceValues`  $\{\langle object\ type\rangle\}$   $\{\langle instance\rangle\}$

`\ShowCollectionInstanceValues`  $\{\langle object\ type\rangle\}$   $\{\langle collection\rangle\}$   $\{\langle instance\rangle\}$

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  $\langle key\rangle$   $\langle choice\rangle$ . 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_toks
```

This token register 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 `cname` 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  $\langle type \rangle$  is equal to  $\langle num \rangle$ , then executes either  $\langle true\ code \rangle$  or generates an error as appropriate.

<code>\xtemplate_execute_if_code_exist:nnT</code>	<code>\xtemplate_execute_if_code_exist:nnT {&lt;type&gt;} {&lt;template code&gt;} {&lt;true code&gt;}</code>
---	--

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

<code>\xtemplate_execute_if_keytype_exist:nT</code> <code>\xtemplate_execute_if_keytype_exist:VT</code>	<code>\xtemplate_execute_if_keytype_exist:nT {&lt;keytype&gt;}</code> <code>{&lt;true code&gt;}</code>
--	---

Tests if  $\langle keytype \rangle$  is a known keytype, then executes either  $\langle true\ code \rangle$  or generates an error as appropriate.

<code>\xtemplate_execute_if_type_exist:nT</code>	<code>\xtemplate_if_type_exist:nT {&lt;type&gt;} {&lt;true code&gt;}</code>
--	---

Tests if template  $\langle type \rangle$  has been created, then executes either  $\langle true\ code \rangle$  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  $\langle template \rangle$  of  $\langle type \rangle$  have been declared (but not necessarily given an implementation), , then executes either  $\langle true\ code \rangle$  or generates an error as appropriate.

### 8.3 Utility functions

<pre>\xtemplate_if_key_value:nT *</pre>	
<pre>\xtemplate_if_key_value:VT *</pre>	<pre>\xtemplate_if_key_value:nT  {(tokens)}  {(true code)}</pre>

Tests if the first token in `<tokens>` is `\KeyValue`.

$\backslash\text{template\_if\_eval\_now:nTF} \star$	$\backslash\text{template\_if\_eval\_now:nTF} \quad \{\langle\textit{tokens}\rangle\}$ $\quad \{\langle\textit{true code}\rangle\} \quad \{\langle\textit{false code}\rangle\}$
--	--

Tests if the first token in `<tokens>` is a marker for evaluating now (`\EvaluateNow`).

	<code>\xtemplate_if_instance_exist:nnnTF</code>	<code>{\type}</code>
		<code>{\collection}</code> <code>{\instance}</code> <code>{\true code}</code>
<code>\xtemplate_if_instance_exist:nnnTF</code>	<code>*</code>	<code>{\false code}</code>

Tests if  $\langle instance \rangle$  of  $\langle type \rangle$  exists for the  $\langle collection \rangle$  given.

<code>\xtemplate_if_use_template:nTF *</code>	<code>\xtemplate_if_use_template:nTF {⟨assignment⟩}  {⟨true code⟩} {⟨false code⟩}</code>
---	--

Tests if assignment begins with `\UseTemplate`.

<code>\xtemplate_store_defaults:n  \xtemplate_store_keytypes:n  \xtemplate_store_restrictions:n  \xtemplate_store_values:n  \xtemplate_store_vars:n</code>	<code>\xtemplate_store_defaults:n {⟨full name⟩}</code>
--	--

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).

<code>\xtemplate_recover_defaults:n  \xtemplate_recover_keytypes:n  \xtemplate_recover_restrictions:n  \xtemplate_recover_values:n  \xtemplate_recover_vars:n</code>	<code>\xtemplate_recover_defaults:n {⟨full name⟩}</code>
--	--

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

<code>\xtemplate_declare_object_type:nn</code>	<code>\xtemplate_declare_object_type:nn {⟨type⟩} {⟨num⟩}</code>
--	---

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

## 8.5 Declaring template keys

<code>\xtemplate_declare_template_keys:nnnn</code>	<code>\xtemplate_declare_template_keys:nnnn {⟨type⟩}  {⟨template⟩} {⟨num⟩} {⟨keyvals⟩}</code>
--	---

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



<code>\xtemplate_parse_keys_elt:n</code> <code>\xtemplate_parse_keys_elt:nn</code>	<code>\xtemplate_parse_keys_elt:nn {&lt;key&gt;} {&lt;value&gt;}</code>
---	---

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

<code>\xtemplate_split_keytype:n</code>	<code>\xtemplate_split_keytype:n {&lt;key&gt;}</code>
---	---

Splits a *<key>* into a key name (stored as `\l_xtemplate_key_tl`) and a keytype (stored as `\l_xtemplate_keytype_tl`).

<code>\xtemplate_split_keytype_arg:n</code> <code>\xtemplate_split_keytype_arg:V</code>	<code>\xtemplate_split_keytype_arg:n {&lt;keytype&gt;}</code>
--	---

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

<code>\xtemplate_store_value_boolean:n</code> <code>\xtemplate_store_value_choice:n</code> <code>\xtemplate_store_value_choice:V</code> <code>\xtemplate_store_value_code:n</code> <code>\xtemplate_store_value_commalist:n</code> <code>\xtemplate_store_value_function:n</code> <code>\xtemplate_store_value_function:n</code> <code>\xtemplate_store_value_instance:n</code> <code>\xtemplate_store_value_tokenlist:n</code> <code>\xtemplate_store_value_integer:n</code> <code>\xtemplate_store_value_length:n</code> <code>\xtemplate_store_value_skip:n</code>	<code>\xtemplate_store_value_boolean:n {&lt;value&gt;}</code>
--	---

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>*.

<code>\xtemplate_store_value_choice_name:n</code>	<code>\xtemplate_store_value_choice_name:n {&lt;value&gt;}</code>
---	---

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

<code>\xtemplate_declare_template_code:nnnnn</code>	<code>\xtemplate_declare_template_code:nnnnn {&lt;type&gt;} {&lt;template&gt;} {&lt;num&gt;} {&lt;keyvals&gt;} {&lt;code&gt;}</code>
---	--

Declares implementation of  $\langle template \rangle$  of  $\langle type \rangle$ , and accepting  $\langle num \rangle$  arguments, with keys implemented as listed in  $\langle keyvals \rangle$  and with  $\langle code \rangle$  to be executed when the  $\langle template \rangle$  is used.

<code>\xtemplate_store_key_implementation:nnn</code>	<code>\xtemplate_store_key_implementation:nnn {&lt;type&gt;} {&lt;template&gt;} {&lt;keyvals&gt;}</code>
--	--

Stores the implementation for keys as specified in  $\langle keyvals \rangle$  for a  $\langle template \rangle$  of  $\langle type \rangle$ .

<code>\xtemplate_parse_vars_elt:n</code> <code>\xtemplate_parse_vars_elt:nn</code>	<code>\xtemplate_parse_vars_elt:nn {&lt;key&gt;} {&lt;variable&gt;}</code>
---	--

Used by the key-value parser to assign a  $\langle variable \rangle$  for each  $\langle key \rangle$  listed.

<code>\xtemplate_store_key_implementation:nnn</code>	<code>\xtemplate_store_key_implementation:nnn {&lt;type&gt;} {&lt;template&gt;} {&lt;keyvals&gt;}</code>
--	--

Stores the implementation for keys as specified in  $\langle keyvals \rangle$  for a  $\langle template \rangle$  of  $\langle type \rangle$ .

<code>\xtemplate_implement_choices:n</code>	<code>\xtemplate_implement_choices:n key-value list</code>
---	--

Master function for turning  $\langle key-value list \rangle$  into a set of choices.

<code>\xtemplate_implement_choice_elt:n</code> <code>\xtemplate_implement_choice_elt:nn</code>	<code>\xtemplate_implement_choice_elt:nn {&lt;choice&gt;} {&lt;code&gt;}</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

<code>\xtemplate_declare_restricted:nnnn</code>	<code>\xtemplate_declare_restricted:nnnn {&lt;type&gt;} {&lt;parent&gt;} {&lt;restricted&gt;} {&lt;keyvals&gt;}</code>
---	--

Creates  $\langle restricted \rangle$  template of  $\langle type \rangle$  based on  $\langle parent \rangle$  by fixing values as listed in  $\langle keyvals \rangle$ .

<code>\xtemplate_edit_defaults:nnn</code>	<code>\xtemplate_edit_defaults:nnn {&lt;type&gt;} {&lt;template&gt;}</code> <code>{&lt;keyvals&gt;}</code>
---	---

Modifies the default values for  $\langle template \rangle$  of  $\langle type \rangle$  as instructed in  $\langle keyvals \rangle$ .

<code>\xtemplate_parse_values:nn</code>	<code>\xtemplate_parse_values:nn {&lt;name&gt;} {&lt;keyvals&gt;}</code>
---	--

Parses  $\langle keyvals \rangle$  for full  $\langle name \rangle$ , finding the value for each key and storing it for later assignment.

<code>\xtemplate_parse_values_elt:n</code> <code>\xtemplate_parse_values_elt:nn</code>	<code>\xtemplate_parse_values_elt:nn {&lt;key&gt;} {&lt;variable&gt;}</code>
---	--

Used by the key-value parser to find  $\langle value \rangle$  to assign to implementation of  $\langle key \rangle$ .

<code>\xtemplate_set_template_eq:nn</code>	<code>\xtemplate_set_template_eq:nn {&lt;copy&gt;} {&lt;parent&gt;}</code>
--	--

Copies all of  $\langle parent \rangle$  template to the  $\langle copy \rangle$ , where both are full names (i.e., a template plus type).

## 8.9 Creating instances

<code>\xtemplate_declare_instance:nnnnn</code>	<code>\xtemplate_declare_instance:nnnnn {&lt;type&gt;} {&lt;template&gt;}</code> <code>{&lt;collection&gt;} {&lt;instance&gt;} {&lt;keyvals&gt;}</code>
--	--

Declares an  $\langle instance \rangle$  (within  $\langle collection \rangle$ ) of  $\langle template \rangle$  of  $\langle type \rangle$ , using  $\langle keyvals \rangle$  to define the instance.

<code>\xtemplate_edit_instance:nnnn</code>	<code>\xtemplate_declare_instance:nnnn {&lt;type&gt;} {&lt;collection&gt;}</code> <code>{&lt;instance&gt;} {&lt;keyvals&gt;}</code>
--	--

Modifies an  $\langle instance \rangle$  (within  $\langle collection \rangle$ ) of  $\langle type \rangle$ , using  $\langle keyvals \rangle$  to modify the instance.

<code>\xtemplate_convert_to_assignments:</code>	<code>\xtemplate_convert_to_assignments:</code>
---	---

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

<code>\xtemplate_find_global:</code>	<code>\xtemplate_find_global:</code>
--------------------------------------	--------------------------------------

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

<code>\xtemplate_assign_boolean:</code> <code>\xtemplate_assign_choice:</code> <code>\xtemplate_assign_code:</code> <code>\xtemplate_assign_code:n</code> <code>\xtemplate_assign_commalist:</code> <code>\xtemplate_assign_function:</code> <code>\xtemplate_assign_instance:</code> <code>\xtemplate_assign_integer:</code> <code>\xtemplate_assign_length:</code> <code>\xtemplate_assign_skip:</code> <code>\xtemplate_assign_tokenlist:</code>	<code>\xtemplate_assign_boolean:</code>
---	---

Convert the given  $\langle keytype \rangle$  of  $\langle key \rangle$  into an assignment to a  $\langle variable \rangle$ .

<code>\xtemplate_assign_variable:N</code>	<code>\xtemplate_assign_variable:N <math>\langle function \rangle</math></code>
---	---

Convert the current contents of `\l_xtemplate_value_tl` into an assignment using  $\langle function \rangle$  to the variable named in `\l_xtemplate_var_tl`.

<code>\xtemplate_key_to_value:</code>	<code>\xtemplate_key_to_value:</code>
---------------------------------------	---------------------------------------

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

## 8.11 Using instances

<code>\xtemplate_use_instance:nn</code>	<code>\xtemplate_use_instance:nn <math>\{\langle type \rangle\}</math> <math>\{\langle instance \rangle\}</math></code>
---	---

Executes code stored for  $\langle instance \rangle$  of  $\langle type \rangle$ , taking account of any active collection.

<code>\xtemplate_use_template:nnn</code>	<code>\xtemplate_use_template:nnn <math>\{\langle type \rangle\}</math> <math>\{\langle template \rangle\}</math> <math>\{\langle settings \rangle\}</math></code>
--	--

Executes code stored for  $\langle template \rangle$  of  $\langle type \rangle$  using  $\langle settings \rangle$ .

<code>\xtemplate_use_collection:nn</code>	<code>\xtemplate_use_collection:nn <math>\{\langle type \rangle\}</math> <math>\{\langle collection \rangle\}</math></code>
---	---

Activates  $\langle collection \rangle$  for instances of  $\{\langle type \rangle\}$ .

`\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_toks`, and therefore executes the assignments stored there.

`\xtemplate_assignments_push:n` `\xtemplate_assignments_push:n {<assignments>}`

Pushes `<assignments>` to `\l_xtemplate_assignment_toks` 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.

```

1 <*package>
2 \ProvidesExplPackage
3   {\filename}{\filedate}{\fileversion}{\filedescription}
4 \RequirePackage{expl3}
5 </package>
6 <*initex | package>

```

### 9.0.1 Variables and constants

\c_xtemplate_code_root_tl	So that literal values are kept to a minimum.
\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	
\c_xtemplate_key_order_tl	A special property name, used to store the order keys are defined in.
\c_xtemplate_keytypes_arg_clist	A list of keytypes which also need additional data (an argument), used to parse the keytype correctly.
\g_xtemplate_object_type_prop	For storing types and the associated number of arguments.
\l_xtemplate_assignments_toks	When creating an instance, the assigned values are collected here. A toks is needed as there can be # tokens.
\l_xtemplate_collection_tl	The current instance collection name is stored here.

<code>\l_xtemplate_collections_prop</code>	Lists current collection in force, indexed by object type.  22 <code>\prop_new:N \l_xtemplate_collections_prop</code>
<code>\l_xtemplate_default_tl</code>	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.  23 <code>\tl_new:N \l_xtemplate_default_tl</code>
<code>\l_xtemplate_error_bool</code>	A flag for errors to be carried forward.  24 <code>\bool_new:N \l_xtemplate_error_bool</code>
<code>\l_xtemplate_global_bool</code>	Used to indicate that assignments should be global.  25 <code>\bool_new:N \l_xtemplate_global_bool</code>
<code>\l_xtemplate_restrict_bool</code>	A flag to indicate that a template is being restricted.  26 <code>\bool_new:N \l_xtemplate_restrict_bool</code>
<code>\l_xtemplate_restricted_clist</code>	A scratch list for restricting templates.  27 <code>\clist_new:N \l_xtemplate_restricted_clist</code>
<code>\l_xtemplate_key_name_tl</code> <code>\l_xtemplate_keytype_tl</code> <code>\l_xtemplate_keytype_arg_tl</code> <code>\l_xtemplate_value_tl</code> <code>\l_xtemplate_var_tl</code>	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.  28 <code>\tl_new:N \l_xtemplate_key_name_tl</code> 29 <code>\tl_new:N \l_xtemplate_keytype_tl</code> 30 <code>\tl_new:N \l_xtemplate_keytype_arg_tl</code> 31 <code>\tl_new:N \l_xtemplate_value_tl</code> 32 <code>\tl_new:N \l_xtemplate_var_tl</code>
<code>\l_xtemplate_key_seq</code>	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 <i>is</i> ordered. This will then be stored in the property list.  33 <code>\seq_new:N \l_xtemplate_key_seq</code>
<code>\l_xtemplate_keytypes_prop</code> <code>\l_xtemplate_values_prop</code> <code>\l_xtemplate_vars_prop</code>	To avoid needing too many difficult-to-follow csname assignments, various scratch token registers are used to build up data, which is then transferred  34 <code>\prop_new:N \l_xtemplate_keytypes_prop</code> 35 <code>\prop_new:N \l_xtemplate_values_prop</code> 36 <code>\prop_new:N \l_xtemplate_vars_prop</code>

<code>\l_xtemplate_tmp_clist</code> <code>\l_xtemplate_tmp_dim</code> <code>\l_xtemplate_tmp_int</code> <code>\l_xtemplate_tmp_skip</code>	<p>For pre-processing the data stored by <code>xtemplate</code>, a number of scratch variables are needed. The assignments are made to these in the first instance, unless evaluation is delayed.</p> <pre> 37 \clist_new:N \l_xtemplate_tmp_clist 38 \dim_new:N \l_xtemplate_tmp_dim 39 \int_new:N \l_xtemplate_tmp_int 40 \skip_new:N \l_xtemplate_tmp_skip </pre>
<code>\l_xtemplate_tmp_tl</code>	<p>A scratch variable for comparisons and so on.</p> <pre> 41 \tl_new:N \l_xtemplate_tmp_tl </pre>
<code>\xtemplate_space:</code>	<p>A space!</p> <pre> 42 \cs_new_eq:NN \xtemplate_space: \iow_space: </pre>

## 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.

<code>\xtemplate_execute_if_arg_agree:nnT</code>	<p>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</p>
--	---

```

43 \cs_new:Npn \xtemplate_execute_if_arg_agree:nnT #1#2#3 {
44   \prop_get:NnN \g_xtemplate_object_type_prop {#1} \l_xtemplate_tmp_tl
45   \intexpr_compare:nTF { #2 = \l_xtemplate_tmp_tl }
46     {#3}
47     {
48       \msg_kernel_error:nnxxx { xtemplate }
49       { argument-number-mismatch } {#1} { \l_xtemplate_tmp_tl } {#2}
50     }
51 }

```

<code>\xtemplate_execute_if_code_exist:nnT</code>	<p>A template is only fully declared if the code has been set up, which can be checked by looking for the template function itself.</p>
---	---

```

52 \cs_new:Npn \xtemplate_execute_if_code_exist:nnT #1#2#3 {
53   \cs_if_exist:cTF { \c_xtemplate_code_root_tl #1 / #2 :w }
54     {#3}
55     {
56       \msg_kernel_error:nnxx { xtemplate } { no-template-code }
57       {#1} {#2}
58     }
59 }

```



ate\_execute\_if\_keytype\_exist:nT The test for valid keytypes looks for a function to set up the key, which is part of the  
 ate\_execute\_if\_keytype\_exist:VF ‘code’ side of the template definition. This avoids having different lists for the two parts  
 of the process.

```

60 \cs_new:Npn \xtemplate_execute_if_keytype_exist:nT #1#2 {
61   \cs_if_exist:cTF { xtemplate_store_value_ #1 :n }
62     {#2}
63     { \msg_kernel_error:nnx { xtemplate } { unknown-keytype } {#1} }
64 }
65 \cs_generate_variant:Nn \xtemplate_execute_if_keytype_exist:nT { V }

```

plate\_execute\_if\_type\_exist:nT To check that a particular object type is valid.

```

66 \cs_new:Npn \xtemplate_execute_if_type_exist:nT #1#2 {
67   \prop_if_in:NnTF \g_xtemplate_object_type_prop {#1}
68     {#2}
69     { \msg_kernel_error:nnx { xtemplate } { unknown-object-type } {#1} }
70 }

```

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.

```

71 \cs_new:Npn \xtemplate_if_keys_exist:nnT #1#2#3 {
72   \cs_if_exist:cTF
73     { g_ \c_xtemplate_keytypes_root_tl #1 / #2 _prop }
74     {#3}
75     {
76       \msg_kernel_error:nnxx { xtemplate } { unknown-template }
77         {#1} {#2}
78     }
79 }

```

\xtemplate\_if\_key\_value:nT Tests for the first token in a string being \KeyValue, where \DelayEvaluation is not  
 \xtemplate\_if\_key\_value:VT important.

```

80 \prg_set_conditional:Nnn \xtemplate_if_key_value:n { T } {
81   \tl_if_eq:noTF { \KeyValue } { \tl_head:w #1 \q_nil } {
82     \prg_return_true:
83   }{
84     \prg_return_false:
85   }
86 }
87 \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.

```

88 \prg_new_conditional:Nnn \xtemplate_if_eval_now:n { TF } {
89   \tl_if_eq:noTF { \EvaluateNow } { \tl_head:w #1 \q_nil } {
90     \prg_return_true:

```

```

91   }{
92     \prg_return_false:
93   }
94 }

```

`\xtemplate_if_instance_exist:nnnTF` Testing for an instance is collection dependent.

```

95 \prg_new_conditional:Nnn \xtemplate_if_instance_exist:nnn { T, F, TF } {
96   \cs_if_exist:cTF { \c_xtemplate_instances_root_tl #1 / #2 / #3 :w } {
97     \prg_return_true:
98   }{
99     \prg_return_false:
100   }
101 }

```

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

```

102 \prg_new_conditional:Nnn \xtemplate_if_use_template:n { TF } {
103   \tl_if_eq:noTF { \UseTemplate } { \tl_head:w #1 \q_nil } {
104     \prg_return_true:
105   }{
106     \prg_return_false:
107   }
108 }

```

### 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` The defaults and keytypes are transferred from the scratch property lists to the ‘proper’ lists for the template being created.

```

\xtemplate_store_defaults:n
\xtemplate_store_keytypes:n
\xtemplate_store_restrictions:n
\xtemplate_store_values:n
\xtemplate_store_vars:n
109 \cs_new:Npn \xtemplate_store_defaults:n #1 {
110   \cs_if_free:cT { g_ \c_xtemplate_defaults_root_tl #1 _prop } {
111     \prop_new:c { g_ \c_xtemplate_defaults_root_tl #1 _prop }
112   }
113   \prop_gset_eq:cN { g_ \c_xtemplate_defaults_root_tl #1 _prop }
114     \l_xtemplate_values_prop
115 }
116 \cs_new:Npn \xtemplate_store_keytypes:n #1 {
117   \cs_if_free:cTF { g_ \c_xtemplate_keytypes_root_tl #1 _prop } {
118     \msg_kernel_info:nnx { xtemplate } { define-template-interface }
119       {#1}
120     \prop_new:c { g_ \c_xtemplate_keytypes_root_tl #1 _prop }
121   }
122   {
123     \msg_kernel_warning:nnx { xtemplate }
124       { redefine-template-interface } {#1}

```

```

125     }
126     \prop_gset_eq:cN { g_ \c_xtemplate_keytypes_root_tl #1 _prop }
127     \l_xtemplate_keytypes_prop
128   }
129   \cs_new:Npn \xtemplate_store_values:n #1 {
130     \cs_if_free:cT { l_ \c_xtemplate_values_root_tl #1 _prop } {
131       \prop_new:c { l_ \c_xtemplate_values_root_tl #1 _prop }
132     }
133     \prop_set_eq:cN { l_ \c_xtemplate_values_root_tl #1 _prop }
134     \l_xtemplate_values_prop
135   }
136   \cs_new:Npn \xtemplate_store_restrictions:n #1 {
137     \clist_gset_eq:cN { g_ \c_xtemplate_restrict_root_tl #1 _clist }
138     \l_xtemplate_restrict_clist
139   }
140   \cs_new:Npn \xtemplate_store_vars:n #1 {
141     \cs_if_free:cTF { g_ \c_xtemplate_vars_root_tl #1 _prop } {
142       \msg_kernel_info:nnx { xtemplate } { define-template-code } {#1}
143       \prop_new:c { g_ \c_xtemplate_vars_root_tl #1 _prop }
144     }
145     {
146       \msg_kernel_warning:nnx { xtemplate } { redefine-template-code }
147       {#1}
148     }
149     \prop_gset_eq:cN { g_ \c_xtemplate_vars_root_tl #1 _prop }
150     \l_xtemplate_vars_prop
151   }

```

\xtemplate\_recover\_defaults: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.

\xtemplate\_recover\_keytypes:n  
\xtemplate\_recover\_restrictions:n  
\xtemplate\_recover\_values:n  
\xtemplate\_recover\_vars:n

```

152   \cs_new:Npn \xtemplate_recover_defaults:n #1 {
153     \prop_set_eq:Nc \l_xtemplate_values_prop
154     { g_ \c_xtemplate_defaults_root_tl #1 _prop }
155   }
156   \cs_new:Npn \xtemplate_recover_keytypes:n #1 {
157     \prop_set_eq:Nc \l_xtemplate_keytypes_prop
158     { g_ \c_xtemplate_keytypes_root_tl #1 _prop }
159   }
160   \cs_new:Npn \xtemplate_recover_restrictions:n #1 {
161     \clist_set_eq:Nc \l_xtemplate_restrict_clist
162     { g_ \c_xtemplate_restrict_root_tl #1 _clist }
163   }
164   \cs_new:Npn \xtemplate_recover_values:n #1 {
165     \prop_set_eq:Nc \l_xtemplate_values_prop
166     { l_ \c_xtemplate_values_root_tl #1 _prop }
167   }
168   \cs_new:Npn \xtemplate_recover_vars:n #1 {
169     \prop_set_eq:Nc \l_xtemplate_vars_prop
170     { g_ \c_xtemplate_vars_root_tl #1 _prop }

```

```
171 }
```

### 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.

```
172 \cs_new:Npn \xtemplate_declare_object_type:nn #1#2 {
173   \int_set:Nn \l_xtemplate_tmp_int {#2}
174   \bool_if:nTF {
175     \intexpr_compare_p:n { #2 > \c_nine } ||
176     \intexpr_compare_p:n { #2 < \c_zero }
177   } {
178     \msg_kernel_error:nxxx { xtemplate } { bad-number-of-arguments }
179     {#1} { \exp_not:V \l_xtemplate_tmp_int }
180   }{
181     \prop_if_in:NnTF \g_xtemplate_object_type_prop {#1}
182     {
183       \msg_kernel_warning:nxxx { xtemplate } { redefine-object-type }
184       {#1} {#2}
185     }
186     {
187       \msg_kernel_info:nxxx { xtemplate } { define-object-type }
188       {#1} {#2}
189     }
190     \prop_gput:NnV \g_xtemplate_object_type_prop {#1}
191     \l_xtemplate_tmp_int
192   }
193 }
```

## 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 off to the storage routine to save the parsed information properly.

```
194 \cs_new:Npn \xtemplate_declare_template_keys:nnnn #1#2#3#4 {
195   \xtemplate_execute_if_type_exist:nT {#1}
196   {
```

```

197     \xtemplate_execute_if_arg_agree:nnT {#1} {#3}
198     {
199         \prop_clear:N \l_xtemplate_values_prop
200         \prop_clear:N \l_xtemplate_keytypes_prop
201         \seq_clear:N \l_xtemplate_key_seq
202         \cs_set_eq:NN \KV_key_no_value_elt:n \xtemplate_parse_keys_elt:n
203         \cs_set_eq:NN \KV_key_value_elt:nn \xtemplate_parse_keys_elt:nn
204         \KV_parse_space_removal_sanitiz:n {#4}
205         \prop_put:NVV \l_xtemplate_keytypes_prop \c_xtemplate_key_order_tl
206     \l_xtemplate_key_seq
207         \xtemplate_store_defaults:n { #1 / #2 }
208         \xtemplate_store_keytypes:n { #1 / #2 }
209     }
210 }
211 }

```

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

```

212 \cs_new:Npn \xtemplate_parse_keys_elt:n #1 {
213   \xtemplate_split_keytype:n {#1}
214   \bool_if:NF \l_xtemplate_error_bool
215   {
216     \xtemplate_execute_if_keytype_exist:VT \l_xtemplate_keytype_tl
217     {
218       \clist_map_function:nN { choice , function , instance }
219       \xtemplate_parse_keys_elt_aux:n
220       \bool_if:NF \l_xtemplate_error_bool
221       {
222         \seq_if_in:NVTF \l_xtemplate_key_seq
223         \l_xtemplate_key_name_tl
224         {
225           \msg_kernel_error:nnx { xtemplate }
226           { duplicate-key-interface }
227           { \l_xtemplate_key_name_tl }
228         }
229         { \xtemplate_parse_keys_elt_aux: }
230       }
231     }
232   }
233 }
234 \cs_new_nopar:Npn \xtemplate_parse_keys_elt_aux:n #1 {
235   \tl_if_eq:VnT \l_xtemplate_keytype_tl {#1} {
236     \tl_if_empty:NT \l_xtemplate_keytype_arg_tl {
237       \msg_kernel_error:nnx { xtemplate }

```

```

238         { keytype-requires-argument } {#1}
239         \bool_set_true:N \l_xtemplate_error_bool
240         \clist_map_break:
241     }
242 }
243 }
244 \cs_new_nopar:Npn \xtemplate_parse_keys_elt_aux: {
245     \tl_set:Nx \l_xtemplate_tmp_tl {
246         \l_xtemplate_keytype_tl
247         \l_xtemplate_keytype_arg_tl
248     }
249     \prop_put:NVV \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
250     \l_xtemplate_tmp_tl
251     \seq_put_right:NV \l_xtemplate_key_seq \l_xtemplate_key_name_tl
252     \tl_if_eq:VnT \l_xtemplate_keytype_tl { choice } {
253         \clist_if_in:NnT \l_xtemplate_keytype_arg_tl { unknown } {
254             \msg_kernel_error:nn { xtemplate } { choice-unknown-reserved }
255         }
256     }
257 }

```

`\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!

```

258 \cs_new:Npn \xtemplate_parse_keys_elt:nn #1#2 {
259     \xtemplate_parse_keys_elt:n {#1}
260     \tl_if_eq:VnTF \l_xtemplate_keytype_tl { choice } {
261         \xtemplate_store_value_choice_name:n {#2}
262     }{
263         \use:c { xtemplate_store_value_ \l_xtemplate_keytype_tl :n } {#2}
264     }
265 }

```

`\xtemplate_split_keytype:n` The keytype and key name should be separated by ‘:’. As the definition might be given inside or outside of a code block, 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.

`\xtemplate_split_keytype_aux:w`

```

266 \group_begin:
267 \char_set_lccode:nn { \@ } { \@: }
268 \char_make_other:N \@
269 \tl_to_lowercase:n {
270     \group_end:
271     \cs_new:Npn \xtemplate_split_keytype:n #1 {
272         \bool_set_false:N \l_xtemplate_error_bool
273         \tl_set:Nn \l_xtemplate_tmpa_tl {#1}
274         \tl_replace_all_in:Nnn \l_xtemplate_tmpa_tl { ~ } { }

```

```

275 \tl_replace_all_in:Nnn \l_xtemplate_tmpa_tl { : } { @ }
276 \tl_if_in:VnTF \l_xtemplate_tmpa_tl { @ } {
277   \exp_after:wN \xtemplate_split_keytype_aux:w \l_xtemplate_tmpa_tl
278   \q_stop
279 }{
280   \bool_set_true:N \l_xtemplate_error_bool
281   \msg_kernel_error:nnx { xtemplate } { no-keytype } {#1}
282 }
283 }
284 \cs_new:Npn \xtemplate_split_keytype_aux:w #1 @ #2 \q_stop {
285   \tl_if_empty:nT {#1} {
286     \msg_kernel_error:nnx { xtemplate } { empty-key-name } { @ #2 }
287   }
288   \tl_set:Nx \l_xtemplate_key_name_tl { \tl_to_str:n {#1} }
289   \xtemplate_split_keytype_arg:n {#2}
290 }
291 }

```

\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).

```

292 \cs_new:Npn \xtemplate_split_keytype_arg:n #1 {
293   \tl_set:Nn \l_xtemplate_keytype_tl {#1}
294   \tl_clear:N \l_xtemplate_keytype_arg_tl
295   \cs_set_nopar:Npn \xtemplate_split_keytype_arg_aux:n ##1 {
296     \tl_if_in:nnT {#1} {##1} {
297       \cs_set:Npn \xtemplate_split_keytype_arg_aux:w
298         #####1 ##1 #####2 \q_stop {
299         \tl_if_empty:nT {#####1} {
300           \tl_set:Nn \l_xtemplate_keytype_tl {##1}
301           \tl_set:Nn \l_xtemplate_keytype_arg_tl {#####2}
302           \clist_map_break:
303         }
304       }
305       \xtemplate_split_keytype_arg_aux:w #1 \q_stop
306     }
307   }
308   \clist_map_function:NN \c_xtemplate_keytypes_arg_clist
309   \xtemplate_split_keytype_arg_aux:n
310 }
311 \cs_generate_variant:Nn \xtemplate_split_keytype_arg:n { V }
312 \cs_new_nopar:Npn \xtemplate_split_keytype_arg_aux:n #1 { }
313 \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`).

```
314 \cs_new:Npn \xtemplate_store_value_boolean:n #1 {
315   \xtemplate_if_eval_now:nTF {#1} {
316     \bool_if:cTF { c_ #1 _bool } {
317       \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl
318         { true }
319     }{
320       \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl
321         { false }
322     }
323   }{
324     \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
325   }
326 }
```

`xtemplate_store_value_choice: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.

`xtemplate_store_value_choice:V`

`xtemplate_store_value_choice_aux:n`

`xtemplate_store_value_choice_aux:V`

`xtemplate_store_value_choice_name:n`

```
327 \cs_new:Npn \xtemplate_store_value_choice:n #1 {
328   \tl_set:Nx \l_xtemplate_tmp_tl
329     { \l_xtemplate_key_name_tl \xtemplate_space: #1 }
330   \prop_if_in:NVTF \l_xtemplate_vars_prop \l_xtemplate_tmp_tl {
331     \xtemplate_store_value_choice_aux:V \l_xtemplate_tmp_tl
332   }{
333     \tl_set:Nx \l_xtemplate_tmp_tl
334       { \l_xtemplate_key_name_tl \xtemplate_space: unknown }
335     \prop_if_in:NVTF \l_xtemplate_vars_prop \l_xtemplate_tmp_tl {
336       \xtemplate_store_value_choice_aux:V \l_xtemplate_tmp_tl
337     }{
338       \prop_get:NVN \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
339         \l_xtemplate_tmp_tl
340     }
341   }
```



```

340     \xtemplate_split_keytype_arg:V \l_xtemplate_tmp_tl
341     \msg_kernel_error:nxxxx { xtemplate } { unknown-choice }
342     {#1}
343     { \l_xtemplate_key_name_tl }
344     { \l_xtemplate_keytype_arg_tl }
345     \prop_gdel:NV \l_xtemplate_values_prop \l_xtemplate_key_name_tl
346   }
347 }
348 }
349 \cs_generate_variant:Nn \xtemplate_store_value_choice:n { V }
350 \cs_new:Npn \xtemplate_store_value_choice_aux:n #1 {
351   \prop_get:NnN \l_xtemplate_vars_prop {#1} \l_xtemplate_tmp_tl
352   \prop_put:NVV \l_xtemplate_values_prop \l_xtemplate_key_name_tl
353     \l_xtemplate_tmp_tl
354 }
355 \cs_generate_variant:Nn \xtemplate_store_value_choice_aux:n { V }
356 \cs_new:Npn \xtemplate_store_value_choice_name:n #1 {
357   \tl_set:Nx \l_xtemplate_tmp_tl
358     { \l_xtemplate_key_name_tl \xtemplate_space: default }
359   \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_tmp_tl {#1}
360 }

```

\xtemplate\_store\_value\_code:n  
\xtemplate\_store\_value\_commalist:n  
\xtemplate\_store\_value\_function:n  
\xtemplate\_store\_value\_instance:n  
\xtemplate\_store\_value\_tokenlist:n

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

```

361 \cs_new:Npn \xtemplate_store_value_code:n #1 {
362   \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
363 }
364 \cs_new:Npn \xtemplate_store_value_commalist:n #1 {
365   \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
366 }
367 \cs_new:Npn \xtemplate_store_value_function:n #1 {
368   \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
369 }
370 \cs_new:Npn \xtemplate_store_value_instance:n #1 {
371   \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
372 }
373 \cs_new:Npn \xtemplate_store_value_tokenlist:n #1 {
374   \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
375 }

```

\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’.

```

376 \cs_new:Npn \xtemplate_store_value_integer:n #1 {
377   \xtemplate_if_eval_now:nTF {#1} {
378     \int_set:Nn \l_xtemplate_tmp_int {#1}
379     \prop_put:NVV \l_xtemplate_values_prop \l_xtemplate_key_name_tl

```

```

380     \l_xtemplate_tmp_int
381   }{
382     \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
383   }
384 }
385 \cs_new:Npn \xtemplate_store_value_length:n #1 {
386   \xtemplate_if_eval_now:nTF {#1} {
387     \dim_set:Nn \l_xtemplate_tmp_dim {#1}
388     \prop_put:NVV \l_xtemplate_values_prop \l_xtemplate_key_name_tl
389       \l_xtemplate_tmp_dim
390   }{
391     \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
392   }
393 }
394 \cs_new:Npn \xtemplate_store_value_skip:n #1 {
395   \xtemplate_if_eval_now:nTF {#1} {
396     \skip_set:Nn \l_xtemplate_tmp_skip {#1}
397     \prop_put:NVV \l_xtemplate_values_prop \l_xtemplate_key_name_tl
398       \l_xtemplate_tmp_skip
399   }{
400     \prop_put:NVn \l_xtemplate_values_prop \l_xtemplate_key_name_tl {#1}
401   }
402 }

```

## 9.2.2 Implementation part of template declaration

`te_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.

```

403 \cs_new:Npn \xtemplate_declare_template_code:nnnnn #1#2#3#4#5 {
404   \xtemplate_execute_if_type_exist:nT {#1}
405   {
406     \xtemplate_execute_if_arg_agree:nnT {#1}{#3}
407     {
408       \xtemplate_if_keys_exist:nnT {#1} {#2}
409       {
410         \xtemplate_store_key_implementation:nnn {#1} {#2} {#4}
411         \cs_generate_from_arg_count:cNnn
412           { \c_xtemplate_code_root_tl #1 / #2 :w } \cs_gset:Npn
413             {#3} {#5}
414       }
415     }
416   }
417 }

```

`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.

```

418 \cs_set:Npn \xtemplate_store_key_implementation:nnn #1#2#3 {
419   \xtemplate_recover_defaults:n { #1 / #2 }
420   \xtemplate_recover_keytypes:n { #1 / #2 }
421   \prop_clear:N \l_xtemplate_vars_prop
422   \cs_set_eq:NN \KV_key_no_value_elt:n \xtemplate_parse_vars_elt:n
423   \cs_set_eq:NN \KV_key_value_elt:nn \xtemplate_parse_vars_elt:nn
424   \KV_parse_space_removal_sanitize:n {#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.

```

425   \xtemplate_store_defaults:n { #1 / #2 }
426   \xtemplate_store_vars:n { #1 / #2 }
427   \clist_clear:N \l_xtemplate_restrict_clist
428   \xtemplate_store_restrictions:n { #1 / #2 }
429   \prop_del:NV \l_xtemplate_keytypes_prop \c_xtemplate_key_order_tl
430   \prop_if_empty:NF \l_xtemplate_keytypes_prop {
431     \prop_map_inline:Nn \l_xtemplate_keytypes_prop {
432       \msg_kernel_error:nnxxx { xtemplate } { key-not-implemented }
433       {##1} {#2} {#1}
434     }
435   }
436 }

```

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

```

437 \cs_new:Npn \xtemplate_parse_vars_elt:n #1 {
438   \msg_kernel_error:nnx { xtemplate } { key-no-variable } {#1}
439 }

```

`\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.

```

440 \cs_new:Npn \xtemplate_parse_vars_elt:nn #1#2 {
441   \tl_set:Nx \l_xtemplate_key_name_tl { \tl_to_str:n {#1} }
442   \tl_replace_all_in:Nnn \l_xtemplate_key_name_tl { ~ } { }
443   \prop_if_in:NVTF \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl {
444     \prop_get:NVN \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
445     \l_xtemplate_keytype_tl
446     \xtemplate_split_keytype_arg:V \l_xtemplate_keytype_tl
447     \xtemplate_parse_vars_elt_aux:n {#2}
448     \prop_del:NV \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
449   }{
450     \msg_kernel_error:nnx { xtemplate } { unknown-key } {#1}

```

```

451 }
452 }

```

\template\_parse\_vars\_elt\_aux:n  
\template\_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.

```

453 \cs_new:Npn \template_parse_vars_elt_aux:n #1 {
454   \tl_if_eq:VnTF \l_template_keytype_tl { choice }
455   {
456     \template_implement_choices:n {#1}
457   }
458   {
459     \tl_if_eq:VnTF \l_template_keytype_tl { code }
460     {
461       \prop_put:NVn \l_template_vars_prop
462         \l_template_key_name_tl {#1}
463     }
464     {
465       \tl_if_eq:noTF {#1} { \tl_head:w #1 \q_nil }
466       {
467         \cs_if_exist:NTF #1
468         {
469           \prop_put:NVn \l_template_vars_prop
470             \l_template_key_name_tl {#1}
471         }
472         {
473           \msg_kernel_error:nnx { xtemplate }
474             { undeclared-variable }
475             { \exp_not:N #1 }
476         }
477       }
478       {
479         \tl_if_in:nnTF {#1} { global }
480         { \template_parse_vars_elt_aux:w #1 \q_nil }
481         {
482           \msg_kernel_error:nnx { xtemplate } { bad-variable }
483             { \exp_not:n {#1} }
484         }
485       }
486     }
487   }
488 }
489 \cs_new:Npn \template_parse_vars_elt_aux:w #1 global #2 \q_nil {
490   \tl_if_empty:nTF {#1}
491   {
492     \tl_if_eq:noTF {#2} { \tl_head:w #2 \q_nil }
493     {
494       \cs_if_exist:NTF #2
495       {

```

```

496         \prop_put:NVn \l_xtemplate_vars_prop
497         \l_xtemplate_key_name_tl { #1 global #2 }
498     }
499     {
500         \msg_kernel_error:nnx { xtemplate }
501         { undeclared-variable }
502         { \exp_not:N #2 }
503     }
504 }
505 {
506     \msg_kernel_error:nnx { xtemplate } { bad-variable }
507     { \exp_not:n { #1 global #2 } }
508 }
509 }
510 {
511     \msg_kernel_error:nnx { xtemplate } { bad-variable }
512     { \exp_not:n { #1 global #2 } }
513 }
514 }

```

`\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.

```

515 \cs_new:Npn \xtemplate_implement_choices:n #1 {
516   \clist_set_eq:NN \l_xtemplate_tmp_clist \l_xtemplate_keytype_arg_tl
517   \cs_set_eq:NN \KV_key_no_value_elt:n \xtemplate_implement_choice_elt:n
518   \cs_set_eq:NN \KV_key_value_elt:nn \xtemplate_implement_choice_elt:nn
519   \prop_put:NVn \l_xtemplate_vars_prop \l_xtemplate_key_name_tl { }
520   \KV_parse_space_removal_sanitize:n {#1}
521   \cs_set_eq:NN \KV_key_no_value_elt:n \xtemplate_parse_vars_elt:n
522   \cs_set_eq:NN \KV_key_value_elt:nn \xtemplate_parse_vars_elt:nn
523   \tl_set:Nx \l_xtemplate_tmp_tl
524     { \l_xtemplate_key_name_tl \xtemplate_space: default }
525   \prop_if_in:NVT \l_xtemplate_values_prop \l_xtemplate_tmp_tl {
526     \prop_get:NVN \l_xtemplate_values_prop \l_xtemplate_tmp_tl
527     \l_xtemplate_tmp_tl
528     \xtemplate_store_value_choice:V \l_xtemplate_tmp_tl
529   }
530   \clist_if_empty:NF \l_xtemplate_tmp_clist {
531     \clist_map_inline:Nn \l_xtemplate_tmp_clist
532     {
533       \msg_kernel_error:nnx { xtemplate } { choice-not-implemented }
534       {##1}
535     }
536   }
537 }

```

`\xtemplate_implement_choice_elt:n` The actual storage of the implementation of a choice is mainly about error checking. The code here ensures that all choices have to have been declared, apart from the special

unknown choice, which must come last. The code for each choice is stored along with the key name in the variables property list.

```

538 \cs_new:Npn \xtemplate_implement_choice_elt:n #1 {
539   \clist_if_empty:NTF \l_xtemplate_tmp_clist {
540     \tl_if_eq:nnF {#1} { unknown } {
541       \prop_get:NVN \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
542         \l_xtemplate_tmp_tl
543       \xtemplate_split_keytype_arg:V \l_xtemplate_tmp_tl
544       \msg_kernel_error:nnxxx { xtemplate } { unknown-choice }
545       {#1}
546       { \l_xtemplate_key_name_tl }
547       { \l_xtemplate_keytype_arg_tl }
548     }
549   }{
550     \clist_if_in:NnTF \l_xtemplate_tmp_clist {#1} {
551       \clist_remove_element:Nn \l_xtemplate_tmp_clist {#1}
552     }{
553       \prop_get:NVN \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl
554         \l_xtemplate_tmp_tl
555       \xtemplate_split_keytype_arg:V \l_xtemplate_tmp_tl
556       \msg_kernel_error:nnxxx { xtemplate } { unknown-choice }
557       {#1}
558       { \l_xtemplate_key_name_tl }
559       { \l_xtemplate_keytype_arg_tl }
560     }
561   }
562 }
563 \cs_new:Npn \xtemplate_implement_choice_elt:nn #1#2 {
564   \xtemplate_implement_choice_elt:n {#1}
565   \tl_set:Nx \l_xtemplate_tmp_tl
566     { \l_xtemplate_key_name_tl \xtemplate_space: #1 }
567   \prop_put:NVN \l_xtemplate_vars_prop \l_xtemplate_tmp_tl {#2}
568 }

```

### 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.

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

```

569 \cs_new:Npn \xtemplate_declare_restricted:nnnn #1#2#3#4 {
570   \xtemplate_if_keys_exist:nnT {#1} {#2}
571   {
572     \xtemplate_set_template_eq:nn { #1 / #3 } { #1 / #2 }
573     \bool_set_true:N \l_xtemplate_restrict_bool
574     \xtemplate_edit_defaults_aux:nnn {#1} {#3} {#4}

```

```

575     }
576 }

```

`\xtemplate_edit_defaults:nnn` Editing the template defaults means getting the values back out of the store, then parsing the list of new values before putting the updated list back into storage. The auxiliary function is used to allow code-sharing with the template-restriction system.

```

577 \cs_new:Npn \xtemplate_edit_defaults:nnn {
578   \bool_set_false:N \l_xtemplate_restrict_bool
579   \xtemplate_edit_defaults_aux:nnn
580 }
581 \cs_new:Npn \xtemplate_edit_defaults_aux:nnn #1#2#3 {
582   \xtemplate_if_keys_exist:nnT {#1} {#2}
583   {
584     \xtemplate_recover_defaults:n { #1 / #2 }
585     \xtemplate_recover_restrictions:n { #1 / #2 }
586     \xtemplate_parse_values:nn { #1 / #2 } {#3}
587     \xtemplate_store_defaults:n { #1 / #2 }
588     \xtemplate_store_restrictions:n { #1 / #2 }
589   }
590 }

```

`\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.

```

591 \cs_new:Npn \xtemplate_parse_values:nn #1#2 {
592   \xtemplate_recover_keytypes:n {#1}
593   \clist_clear:N \l_xtemplate_restricted_clist
594   \cs_set_eq:NN \KV_key_no_value_elt:n \xtemplate_parse_values_elt:n
595   \cs_set_eq:NN \KV_key_value_elt:nn \xtemplate_parse_values_elt:nn
596   \KV_parse_space_removal_sanitize:n {#2}
597 }

```

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

```

598 \cs_new:Npn \xtemplate_parse_values_elt:n #1 {
599   \bool_set_true:N \l_xtemplate_error_bool
600   \msg_kernel_error:nxx { xtemplate } { key-no-value } {#1}
601 }

```

`\xtemplate_parse_values_elt:nn` 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.

```

602 \cs_new:Npn \xtemplate_parse_values_elt:nn #1#2 {
603   \tl_set:Nx \l_xtemplate_key_name_tl { \tl_to_str:n {#1} }
604   \tl_replace_all_in:Nnn \l_xtemplate_key_name_tl { ~ } { }
605   \prop_if_in:NVTF \l_xtemplate_keytypes_prop \l_xtemplate_key_name_tl {
606     \bool_if:NTF \l_xtemplate_restrict_bool {

```

```

607     \clist_if_in:NVF \l_xtemplate_restricted_clist
608     \l_xtemplate_key_name_tl {
609     \xtemplate_parse_values_elt_aux:Vn \l_xtemplate_key_name_tl {#2}
610   }
611   }{
612     \xtemplate_parse_values_elt_aux:Vn \l_xtemplate_key_name_tl {#2}
613   }
614   }{
615     \msg_kernel_error:nnx { xtemplate } { unknown-key }
616     { \l_xtemplate_key_name_tl }
617   }
618 }
619 \cs_new:Npn \xtemplate_parse_values_elt_aux:nn #1#2 {
620   \clist_put_right:Nn \l_xtemplate_restricted_clist {#1}
621   \prop_get:NnN \l_xtemplate_keytypes_prop {#1} \l_xtemplate_tmp_tl
622   \xtemplate_split_keytype_arg:V \l_xtemplate_tmp_tl
623   \use:c { xtemplate_store_value_ \l_xtemplate_keytype_tl :n } {#2}
624 }
625 \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.

```

626 \cs_new:Npn \xtemplate_set_template_eq:nn #1#2 {
627   \xtemplate_recover_defaults:n {#2}
628   \xtemplate_store_defaults:n {#1}
629   \xtemplate_recover_keytypes:n {#2}
630   \xtemplate_store_keytypes:n {#1}
631   \xtemplate_recover_vars:n {#2}
632   \xtemplate_store_vars:n {#1}
633   \cs_gset_eq:cc { \c_xtemplate_code_root_tl #1 :w }
634   { \c_xtemplate_code_root_tl #2 :w }
635 }

```

## 9.2.4 Creating instances of templates

`\xtemplate_declare_instance: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.

`\xtemplate_declare_instance_aux:nnnnn`

```

636 \cs_new:Npn \xtemplate_declare_instance:nnnnn #1#2#3#4#5 {
637   \xtemplate_execute_if_code_exist:nnT {#1} {#2}
638   {
639     \xtemplate_recover_defaults:n { #1 / #2 }
640     \xtemplate_recover_vars:n { #1 / #2 }

```



```

641     \xtemplate_declare_instance_aux:nnnnn {#1} {#2} {#3} {#4} {#5}
642   }
643 }
644 \cs_new:Npn \xtemplate_declare_instance_aux:nnnnn #1#2#3#4#5 {
645   \bool_set_false:N \l_xtemplate_error_bool
646   \xtemplate_parse_values:nn { #1 / #2 } {#5}
647   \bool_if:NF \l_xtemplate_error_bool {
648     \prop_put:Nnn \l_xtemplate_values_prop { from~template } {#2}
649     \xtemplate_store_values:n { #1 / #3 / #4 }
650     \xtemplate_convert_to_assignments:
651     \cs_set:cpx { \c_xtemplate_instances_root_tl #1 / #3 / #4 :w } {
652       \exp_not:N \xtemplate_assignments_push:n {
653         \exp_not:V \l_xtemplate_assignments_toks
654       }
655       \exp_not:c { \c_xtemplate_code_root_tl #1 / #2 :w }
656     }
657     \xtemplate_if_instance_exist:nnnF {#1} { } {#4} {
658       \cs_set_eq:cc
659       { \c_xtemplate_instances_root_tl #1 / / #4 :w }
660       { \c_xtemplate_instances_root_tl #1 / #3 / #4 :w }
661     }
662   }
663 }

```

```

\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.

```

664 \cs_new:Npn \xtemplate_edit_instance:nnnn #1#2#3 {
665   \xtemplate_if_instance_exist:nnnTF {#1} {#2} {#3}
666   {
667     \xtemplate_recover_values:n { #1 / #2 / #3 }
668     \prop_get:Nnn \l_xtemplate_values_prop { from~template }
669     \l_xtemplate_tmp_tl
670     \xtemplate_edit_instance_aux:nVnnn {#1} \l_xtemplate_tmp_tl
671     {#2} {#3}
672   }
673   {
674     \msg_kernel_error:nnxx { xtemplate } { unknown~instance }
675     {#1} {#3}
676   }
677 }
678 \cs_new:Npn \xtemplate_edit_instance_aux:nnnnn #1#2 {
679   \xtemplate_recover_vars:n { #1 / #2 }
680   \xtemplate_declare_instance_aux:nnnnn {#1} {#2}
681 }
682 \cs_generate_variant:Nn \xtemplate_edit_instance_aux:nnnnn { nV }

```

```

\xtemplate_convert_to_assignments:
\xtemplate_convert_to_assignments_aux:n
\xtemplate_convert_to_assignments_aux:nn
\xtemplate_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”.

```

683 \cs_new_nopar:Npn \xtemplate_convert_to_assignments: {
684   \toks_clear:N \l_xtemplate_assignments_toks
685   \prop_get:NVN \l_xtemplate_keytypes_prop \c_xtemplate_key_order_tl
686   \l_xtemplate_key_seq
687   \seq_map_function:NN \l_xtemplate_key_seq
688   \xtemplate_convert_to_assignments_aux:n
689 }
690 \cs_new:Npn \xtemplate_convert_to_assignments_aux:n #1 {
691   \prop_get:NnN \l_xtemplate_keytypes_prop {#1} \l_xtemplate_tmp_tl
692   \xtemplate_convert_to_assignments_aux:nV {#1} \l_xtemplate_tmp_tl
693 }

```

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.

```

694 \cs_new:Npn \xtemplate_convert_to_assignments_aux:nn #1#2 {
695   \prop_if_in:NnT \l_xtemplate_values_prop {#1} {
696     \prop_if_in:NnTF \l_xtemplate_vars_prop {#1} {
697       \prop_get:NnN \l_xtemplate_values_prop {#1} \l_xtemplate_value_tl
698       \prop_get:NnN \l_xtemplate_vars_prop {#1} \l_xtemplate_var_tl
699       \xtemplate_split_keytype_arg:n {#2}
700       \tl_if_eq:VnF \l_xtemplate_keytype_tl { choice } {
701         \tl_if_eq:VnF \l_xtemplate_keytype_tl { code } {
702           \xtemplate_find_global:
703         }
704       }
705       \use:c { xtemplate_assign_ \l_xtemplate_keytype_tl : }
706     }{
707       \msg_kernel_error:nxx { xtemplate } { unknown-attribute } {#1}
708     }
709   }
710 }
711 \cs_generate_variant:Nn \xtemplate_convert_to_assignments_aux:nn { nV }

```

\xtemplate\_find\_global: Global assignments should have the phrase “global” at the front. This is pretty easy to  
\xtemplate\_find\_global\_aux:w find: no other error checking, though.

```

712 \cs_new_nopar:Npn \xtemplate_find_global: {
713   \bool_set_false:N \l_xtemplate_global_bool
714   \tl_if_in:VnT \l_xtemplate_var_tl { global } {
715     \exp_after:wN \xtemplate_find_global_aux:w \l_xtemplate_var_tl \q_stop
716   }
717 }
718 \cs_new:Npn \xtemplate_find_global_aux:w #1 global #2 \q_stop {
719   \tl_set:Nn \l_xtemplate_var_tl {#2}

```

```

720 \bool_set_true:N \l_xtemplate_global_bool
721 }

```

### 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.

```

722 \cs_new:Npn \xtemplate_use_template:nnn #1#2#3 {
723   \xtemplate_recover_defaults:n { #1 / #2 }
724   \xtemplate_recover_vars:n { #1 / #2 }
725   \xtemplate_parse_values:nn { #1 / #2 } {#3}
726   \xtemplate_convert_to_assignments:
727   \use:c { \c_xtemplate_code_root_tl #1 / #2 :w }
728 }

```

#### 9.3.1 Assigning values to variables

`\xtemplate_assign_boolean:` Setting a Boolean value is slightly different to everything else as the value can be used to work out which `set` function to call. As long as there is no need to recover things from another variable, everything is pretty easy.

```

729 \cs_new_nopar:Npn \xtemplate_assign_boolean: {
730   \bool_if:NTF \l_xtemplate_global_bool {
731     \xtemplate_assign_boolean_aux:n { bool_gset }
732   }{
733     \xtemplate_assign_boolean_aux:n { bool_set }
734   }
735 }
736 \cs_new_nopar:Npn \xtemplate_assign_boolean_aux:n #1 {
737   \xtemplate_if_key_value:VT \l_xtemplate_value_tl {
738     \xtemplate_key_to_value:
739   }
740   \toks_put_left:Nx \l_xtemplate_assignments_toks {
741     \exp_not:c { #1 _ \l_xtemplate_value_tl :N }
742     \exp_not:V \l_xtemplate_var_tl
743   }
744 }

```

`\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.

```

745 \cs_new_nopar:Npn \xtemplate_assign_choice: {
746   \toks_put_left:NV \l_xtemplate_assignments_toks \l_xtemplate_value_tl
747 }

```

`\xtemplate_assign_code:` Assigning general code to a key needs a scratch function to be created and run when `\xtemplate_assign_code:n` `\AssignTemplateKeys` is called. So the appropriate definition then use is created in the toks.

```

748 \cs_new_nopar:Npn \xtemplate_assign_code: {
749   \toks_put_left:Nx \l_xtemplate_assignments_toks {
750     \exp_not:N \cs_set:Npn \exp_not:N \xtemplate_assign_code:n ##1 {
751       \exp_not:V \l_xtemplate_var_tl
752     }
753     \exp_not:N \xtemplate_assign_code:n
754     { \exp_not:V \l_xtemplate_value_tl }
755   }
756 }
757 \cs_new:Npn \xtemplate_assign_code:n #1 { }
```

`\xtemplate_assign_function:` For functions, the toks needs to have a cs-generation step created. This looks a bit messy but is only actually one function.

`\xtemplate_assign_function_aux:N`

```

758 \cs_new_nopar:Npn \xtemplate_assign_function: {
759   \bool_if:NTF \l_xtemplate_global_bool {
760     \xtemplate_assign_function_aux:N \cs_gset:Npn
761   }{
762     \xtemplate_assign_function_aux:N \cs_set:Npn
763   }
764 }
765 \cs_new_nopar:Npn \xtemplate_assign_function_aux:N #1 {
766   \toks_put_left:Nx \l_xtemplate_assignments_toks {
767     \exp_not:N \cs_generate_from_arg_count:NNnn
768     \exp_not:V \l_xtemplate_var_tl
769     \exp_not:N #1
770     { \exp_not:V \l_xtemplate_keytype_arg_tl }
771     { \exp_not:V \l_xtemplate_value_tl }
772   }
773 }
```

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

`\xtemplate_assign_instance_aux:N`

```

774 \cs_new_nopar:Npn \xtemplate_assign_instance: {
775   \bool_if:NTF \l_xtemplate_global_bool {
776     \xtemplate_assign_instance_aux:N \cs_gset:Npn
777   }{
778     \xtemplate_assign_instance_aux:N \cs_set:Npn
779   }
780 }
781 \cs_new_nopar:Npn \xtemplate_assign_instance_aux:N #1 {
782   \toks_put_left:Nx \l_xtemplate_assignments_toks {
783     \exp_not:N #1 \exp_not:V \l_xtemplate_var_tl {
784       \exp_not:N \xtemplate_use_instance:nn
```

```

785         { \exp_not:V \l_xtemplate_keytype_arg_tl }
786         { \exp_not:V \l_xtemplate_value_tl }
787     }
788 }
789 }

```

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

```

\xtemplate_assign_length:
\xtemplate_assign_skip:
790 \cs_new_nopar:Npn \xtemplate_assign_integer: {
791   \bool_if:NTF \l_xtemplate_global_bool {
792     \xtemplate_assign_variable:N \int_gset:Nn
793   }{
794     \xtemplate_assign_variable:N \int_set:Nn
795   }
796 }
797 \cs_new_nopar:Npn \xtemplate_assign_length: {
798   \bool_if:NTF \l_xtemplate_global_bool {
799     \xtemplate_assign_variable:N \dim_gset:Nn
800   }{
801     \xtemplate_assign_variable:N \dim_set:Nn
802   }
803 }
804 \cs_new_nopar:Npn \xtemplate_assign_skip: {
805   \bool_if:NTF \l_xtemplate_global_bool {
806     \xtemplate_assign_variable:N \skip_gset:Nn
807   }{
808     \xtemplate_assign_variable:N \skip_set:Nn
809   }
810 }

```

`\xtemplate_assign_tokenlist:` 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.

```

\xtemplate_assign_tokenlist_aux:N
811 \cs_new_nopar:Npn \xtemplate_assign_tokenlist: {
812   \bool_if:NTF \l_xtemplate_global_bool {
813     \xtemplate_assign_tokenlist_aux:N \tl_gset:Nn
814   }{
815     \xtemplate_assign_tokenlist_aux:N \tl_set:Nn
816   }
817 }
818 \cs_new_eq:NN \xtemplate_assign_commalist:
819   \xtemplate_assign_tokenlist:
820 \cs_new_nopar:Npn \xtemplate_assign_tokenlist_aux:N #1 {
821   \toks_put_left:Nx \l_xtemplate_assignments_toks {
822     \exp_not:N #1 \exp_not:V \l_xtemplate_var_tl
823     { \exp_not:V \l_xtemplate_value_tl }
824   }
825 }

```

`\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.

```

826 \cs_new_nopar:Npn \xtemplate_assign_variable:N #1 {
827   \xtemplate_if_key_value:VT \l_xtemplate_value_tl {
828     \xtemplate_key_to_value:
829   }
830   \toks_put_left:Nx \l_xtemplate_assignments_toks {
831     \exp_not:N #1 \exp_not:V \l_xtemplate_var_tl
832     { \exp_not:V \l_xtemplate_value_tl }
833   }
834 }

```

`\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.

```

835 \cs_new_nopar:Npn \xtemplate_key_to_value: {
836   \exp_after:wN \xtemplate_key_to_value_aux:w \l_xtemplate_value_tl
837 }
838 \cs_new:Npn \xtemplate_key_to_value_aux:w \KeyValue #1 {
839   \tl_set:Nx \l_xtemplate_tmp_tl { \tl_to_str:n {#1} }
840   \tl_replace_all_in:Nnn \l_xtemplate_key_name_tl { ~ } { }
841   \prop_if_in:NVTF \l_xtemplate_vars_prop \l_xtemplate_tmp_tl {
842     \prop_get:NVN \l_xtemplate_vars_prop \l_xtemplate_tmp_tl
843     \l_xtemplate_value_tl
844   }{
845     \msg_kernel_error:nxx { xtemplate } { unknown-attribute }
846     { \l_xtemplate_tmp_tl }
847   }
848 }

```

### 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.

```

849 \cs_new:Npn \xtemplate_use_instance:nn #1#2 {
850   \xtemplate_if_use_template:nTF {#2} {
851     \xtemplate_use_instance_aux:nNnnn {#1} #2
852   }{
853     \xtemplate_use_instance_aux:nn {#1} {#2}
854   }
855 }

```

```

856 \cs_new:Npn \xtemplate_use_instance_aux:nNnnn #1#2#3#4#5 {
857   \tl_if_eq:nnTF {#1} {#3} {
858     \xtemplate_use_template:nnn {#3} {#4} {#5}
859   }{
860     \msg_kernel_error:nxxx { xtemplate } { type-mismatch } {#1} {#3}
861   }
862 }
863 \cs_new:Npn \xtemplate_use_instance_aux:nn #1#2 {
864   \xtemplate_get_collection:n {#1}
865   \xtemplate_if_instance_exist:nnnTF
866   {#1} { \l_xtemplate_collection_tl } {#2} {
867     \use:c {
868       \c_xtemplate_instances_root_tl #1 / \l_xtemplate_collection_tl /#2
869       :w
870     }
871   }{
872     \xtemplate_if_instance_exist:nnnTF {#1} { } {#2} {
873       \use:c { \c_xtemplate_instances_root_tl #1 / / #2 :w }
874     }{
875       \msg_kernel_error:nxxx { xtemplate } { unknown-instance }
876       {#1} {#2}
877     }
878   }
879 }

```

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

```

880 \cs_new:Npn \xtemplate_use_collection:nn #1#2 {
881   \prop_put:Nnn \l_xtemplate_collections_prop {#1} {#2}
882 }

```

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

```

883 \cs_new:Npn \xtemplate_get_collection:n #1 {
884   \prop_if_in:NnTF \l_xtemplate_collections_prop {#1} {
885     \prop_get:NnN \l_xtemplate_collections_prop {#1}
886     \l_xtemplate_collection_tl
887   }{
888     \tl_clear:N \l_xtemplate_collection_tl
889   }
890 }

```

### 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.

```

891 \cs_new_nopar:Npn \xtemplate_assignments_pop: {
892   \toks_use:N \l_xtemplate_assignments_toks
893 }

```

\xtemplate\_assignments\_push:n Here, the assignments are stored for later use.

```

894 \cs_new:Npn \xtemplate_assignments_push:n #1 {
895   \toks_set:Nn \l_xtemplate_assignments_toks {#1}
896 }

```

### 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.

```

897 \cs_new:Npn \xtemplate_show_code:nn #1#2 {
898   \xtemplate_execute_if_code_exist:nnT {#1} {#2}
899   { \cs_show:c { \c_xtemplate_code_root_tl #1 / #2 :w } }
900 }

```

\xtemplate\_show\_defaults:nn Showing the internal data is a case of getting the appropriate property list back, then displaying the scratch variable.

\xtemplate\_show\_keytypes:nn  
\xtemplate\_show\_values:nnn  
\xtemplate\_show\_vars:nn

```

901 \cs_new:Npn \xtemplate_show_defaults:nn #1#2 {
902   \xtemplate_if_keys_exist:nnT {#1} {#2}
903   {
904     \xtemplate_recover_defaults:n { #1 / #2 }
905     \prop_display:N \l_xtemplate_values_prop
906   }
907 }
908 \cs_new:Npn \xtemplate_show_keytypes:nn #1#2 {
909   \xtemplate_if_keys_exist:nnT {#1} {#2}
910   {
911     \xtemplate_recover_keytypes:n { #1 / #2 }
912     \prop_display:N \l_xtemplate_keytypes_prop
913   }
914 }
915 \cs_new:Npn \xtemplate_show_values:nnn #1#2#3 {
916   \xtemplate_if_instance_exist:nnnTF {#1} {#2} {#3}
917   {
918     \xtemplate_recover_values:n { #1 / #2 / #3 }
919     \prop_display:N \l_xtemplate_values_prop
920   }
921   {
922     \msg_kernel_error:nnxx { xtemplate } { unknown-instance }
923     {#1} {#2}
924   }
925 }
926 \cs_new:Npn \xtemplate_show_vars:nn #1#2 {
927   \xtemplate_execute_if_code_exist:nnT {#1} {#2}

```



```

928 {
929     \xtemplate_recover_vars:n { #1 / #2 }
930     \prop_display:N \l_xtemplate_vars_prop
931 }
932 }

```

### 9.3.5 Messages

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

```

933 \msg_kernel_new:nnnn { xtemplate } { argument-number-mismatch }
934 {Object type ‘#1’ takes ‘#2’ not ‘#3’ argument(s).}
935 {%
936     Objects of type ‘#1’ require ‘#2’ arguments.\\%
937     You have tried to make a template for ‘#1’\\%
938     with ‘#3’ arguments, which is not possible:\\%
939     the number of arguments must agree.%
940 }
941 \msg_kernel_new:nnnn { xtemplate } { bad-number-of-arguments }
942 {%
943     Impossible number of arguments declared for\\%
944     object type ‘#1’.%
945 }
946 {%
947     An object may accept between 0 and 9 arguments.\\%
948     You asked to use #2 arguments: this is not supported.%
949 }
950 \msg_kernel_new:nnnn { xtemplate } { bad-variable }
951 {Incorrect variable description \msg_line_context:.}
952 {%
953     The argument ‘#1’\\%
954     is not of the form ‘<variable>’ or ‘global <variable>’.\\%
955     It must be given in one of these formats to be used in a template.%
956 }
957 \msg_kernel_new:nnnn { xtemplate } { choice-not-implemented }
958 {The choice ‘#1’ has no implementation \msg_line_context:.}
959 {%
960     Each choice listed in the interface for a template must\\%
961     have an implementation.\\%
962     I did not find an implementation for ‘#1’.%
963 }
964 \msg_kernel_new:nnnn { xtemplate } { choice-unknown-reserved }
965 {The choice ‘unknown’ cannot be given for a template key.}
966 {%
967     The ‘unknown’ choice is automatically checked by template when\\%
968     a choice key is given with an unknown value. It should not be\\%
969     included in the list of named choices for a key, and is always\\%
970     given last in the implementation of choices.%
971 }

```

```

972 \msg_kernel_new:nnnn { xtemplate } { duplicate-key-interface }
973 {Key '#1' appears twice in interface definition \msg_line_context:..}
974 {%
975     Each key can only have one interface declared in a template.\\%
976     I found two interfaces for '#1': probably a typing error.%
977 }
978 \msg_kernel_new:nnnn { xtemplate } { empty-key-name }
979 {No key name found in '#1' \msg_line_context:..}
980 {%
981     A template key name and type is given in the form:\\%
982     \msg_space: <name> : <type> \\%
983     Parsing your input, I found a ':' but nothing before
984     it!%
985 }
986 \msg_kernel_new:nnnn { xtemplate } { key-no-value }
987 {The key '#1' has no value \msg_line_context:..}
988 {%
989     When creating an instance of a template\\%
990     every key listed must include a value:
991     \msg_two_spaces: <key> = <value>
992 }
993 \msg_kernel_new:nnnn { xtemplate } { key-no-variable }
994 {The key '#1' requires implementation details \msg_line_context:..}
995 {%
996     When creating template code using
997     \token_to_str:N \DeclareTemplateCode,\\%
998     each key name must have an associated implementation.\\%
999     This should be given after an '=' sign: I did not find one.%
1000 }
1001 \msg_kernel_new:nnnn { xtemplate } { key-not-implemented }
1002 {Key '#1' has no implementation \msg_line_context:..}
1003 {%
1004     The definition of key implementations for template '#2'\\%
1005     of object type '#3' does not include any details for key '#1'.\\%
1006     The key was declared in the interface definition,\\%
1007     and so an implementation is required.%
1008 }
1009 \msg_kernel_new:nnnn { xtemplate } { keytype-requires-argument }
1010 {The keytype '#1' requires an argument \msg_line_context:..}
1011 {%
1012     You should have put:\\%
1013     \msg_two_spaces: <key-name> : #1
1014     \token_to_str:N { <argument> \token_to_str:N } \\%
1015     but I did not find an <argument>.%
1016 }
1017 \msg_kernel_new:nnnn { xtemplate } { no-keytype }
1018 {The key '#1' is missing a keytype \msg_line_context:..}
1019 {%
1020     You should have put:\\%
1021     \msg_two_spaces: #1 : <keytype>\\%

```

```

1022     but I did not find an <keytype>.%
1023 }
1024 \msg_kernel_new:nnnn { xtemplate } { no-template-code }
1025 {%
1026     The template '#2' of type '#1' is unknown\\%
1027     or has no implementation.%
1028 }
1029 {%
1030     You need to declare a template with
1031     \token_to_str:N \DeclareTemplateInterface \\%
1032     and its code with
1033     \token_to_str:N \DeclareTemplateCode \msg_space:
1034     prior to using it.%
1035 }
1036 \msg_kernel_new:nnnn { xtemplate } { type-mismatch }
1037 {Object types '#1' and '#2' do not agree.}
1038 {%
1039     You are trying to use a template directly with
1040     \token_to_str:N \UseInstance \\%
1041     (or a similar function), but the object types do not match.%
1042 }
1043 \msg_kernel_new:nnnn { xtemplate } { undeclared-variable }
1044 {The variable #1has not been declared.}
1045 {%
1046     Before declaring an instance, all of the required variables\\%
1047     used in template keys must be created.%
1048 }
1049 \msg_kernel_new:nnnn { xtemplate } { unknown-attribute }
1050 {The template attribute '#1' is unknown.}
1051 {%
1052     There is a definition in the current template reading\\%
1053     \token_to_str:N \KeyValue
1054     \token_to_str:N { #1 \token_to_str:N }\\%
1055     but there is no key called '#1'.%
1056 }
1057 \msg_kernel_new:nnnn { xtemplate } { unknown-choice }
1058 {The choice '#1' was not declared for key '#2' \msg_line_context:.}
1059 {%
1060     The key '#2' takes a fixed number of choices:\\%
1061     \clist_map_function:NN #3 \xtemplate_unknown_choice_aux:n
1062     and this list does not include '#1'.%
1063 }
1064 \cs_new:Npn \xtemplate_unknown_choice_aux:n #1 { -- #1 ;\\}
1065 \msg_kernel_new:nnnn { xtemplate } { unknown-keytype }
1066 {The keytype '#1' is unknown \msg_line_context:.}
1067 {%
1068     Valid keytypes are:\\%
1069     - boolean;\\%
1070     - choice;\\%
1071     - code;\\%

```

```

1072     - commalist;\%
1073     - function;\%
1074     - instance;\%
1075     - integer;\%
1076     - length;\%
1077     - skip;\%
1078     - tokenlist.%
1079 }
1080 \msg_kernel_new:nnnn { xtemplate } { unknown-object-type }
1081 {The object type '#1' is unknown \msg_line_context:..}
1082 {%
1083     An object type needs to be declared with
1084     \token_to_str:N \DeclareObjectType \msg_space:
1085     prior to using it.%
1086 }
1087 \msg_kernel_new:nnnn { xtemplate } { unknown-instance }
1088 {The instance '#2' of type '#1' is unknown.}
1089 {%
1090     You have asked to use an instance '#2',\%
1091     but this has not been created.%
1092 }
1093 \msg_kernel_new:nnnn { xtemplate } { unknown-key }
1094 {Unknown template key '#1' \msg_line_context:..}
1095 {%
1096     The key '#1' was not declared in the interface\%
1097     for the current template.\%
1098     Probably you have misspelled the key name.%
1099 }
1100 \msg_kernel_new:nnnn { xtemplate } { unknown-template }
1101 {The template '#2' of type '#1' is unknown \msg_line_context:..}
1102 {%
1103     No interface has been declared for a template\%
1104     '#2' of object type '#1'.%
1105 }

```

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

```

1106 \msg_kernel_new:nnn { xtemplate } { define-template-code }
1107 {Defining template code for '#1' \msg_line_context:..}
1108 \msg_kernel_new:nnn { xtemplate } { define-template-interface }
1109 {Defining template interface for '#1' \msg_line_context:..}
1110 \msg_kernel_new:nnn { xtemplate } { define-object-type }
1111 {Defining object type '#1' with #2 argument(s) \msg_line_context:..}
1112 \msg_kernel_new:nnn { xtemplate } { redefine-template-code }
1113 {Redefining template code for '#1' \msg_line_context:..}
1114 \msg_kernel_new:nnn { xtemplate } { redefine-template-interface }
1115 {Redefining template interface for '#1' \msg_line_context:..}
1116 \msg_kernel_new:nnn { xtemplate } { redefine-object-type }
1117 {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 xpase approach only the appropriate arguments are long.

<pre> \DeclareObjectType \DeclareTemplateInterface \DeclareTemplateCode \DeclareRestrictedTemplate \EditTemplateDefaults \DeclareInstance \DeclareCollectionInstance \EditInstance \EditCollectionInstance \UseTemplate \UseInstance \UseCollection </pre>	<p>All simple translations, with the appropriate long/short argument filtering.</p> <pre> 1118 \cs_new_protected_nopar:Npn \DeclareObjectType #1#2 { 1119   \xtemplate_declare_object_type:nn {#1} {#2} 1120 } 1121 \cs_new_protected:Npn \DeclareTemplateInterface #1#2#3#4 { 1122   \xtemplate_declare_template_keys:nnnn {#1} {#2} {#3} {#4} 1123 } 1124 \cs_new_protected:Npn \DeclareTemplateCode #1#2#3#4#5 { 1125   \xtemplate_declare_template_code:nnnnn {#1} {#2} {#3} {#4} {#5} 1126 } 1127 \cs_new_protected:Npn \DeclareRestrictedTemplate #1#2#3#4 { 1128   \xtemplate_declare_restricted:nnnn {#1} {#2} {#3} {#4} 1129 } 1130 \cs_new_protected:Npn \DeclareInstance #1#2#3#4 { 1131   \xtemplate_declare_instance:nnnnn {#1} {#3} { } {#2} {#4} 1132 } 1133 \cs_new_protected:Npn \DeclareCollectionInstance #1#2#3#4#5 { 1134   \xtemplate_declare_instance:nnnnn {#2} {#4} {#1} {#3} {#5} 1135 } 1136 \cs_new_protected:Npn \EditTemplateDefaults #1#2#3 { 1137   \xtemplate_edit_defaults:nnn {#1} {#2} {#3} 1138 } 1139 \cs_new_protected:Npn \EditInstance #1#2#3 { 1140   \xtemplate_edit_instance:nnnn {#1} { } {#2} {#3} 1141 } 1142 \cs_new_protected:Npn \EditCollectionInstance #1#2#3#4 { 1143   \xtemplate_edit_instance:nnnn {#2} {#1} {#3} {#4} 1144 } 1145 \cs_new_protected_nopar:Npn \UseTemplate #1#2#3 { 1146   \xtemplate_use_template:nnn {#1} {#2} {#3} 1147 } 1148 \cs_new_protected_nopar:Npn \UseInstance #1#2 { 1149   \xtemplate_use_instance:nn {#1} {#2} 1150 } 1151 \cs_new_protected_nopar:Npn \UseCollection #1#2 { 1152   \xtemplate_use_collection:nn {#1} {#2} 1153 } </pre>
--	--

<pre> \ShowTemplateCode \ShowTemplateDefaults \ShowTemplateKeytypes \ShowTemplateVariables \ShowInstanceValues \ShowCollectionInstanceValues </pre>	<p>The show functions are again just translation.</p> <pre> 1154 \cs_new_protected_nopar:Npn \ShowTemplateCode #1#2 { 1155   \xtemplate_show_code:nn {#1} {#2} 1156 } </pre>
---	--

```

1157 \cs_new_protected_nopar:Npn \ShowTemplateDefaults #1#2 {
1158   \xtemplate_show_defaults:nn {#1} {#2}
1159 }
1160 \cs_new_protected_nopar:Npn \ShowTemplateKeytypes #1#2 {
1161   \xtemplate_show_keytypes:nn {#1} {#2}
1162 }
1163 \cs_new_protected_nopar:Npn \ShowTemplateVariables #1#2 {
1164   \xtemplate_show_vars:nn {#1} {#2}
1165 }
1166 \cs_new_protected_nopar:Npn \ShowInstanceValues #1#2 {
1167   \xtemplate_show_values:nnn {#1} { } {#2}
1168 }
1169 \cs_new_protected_nopar:Npn \ShowCollectionInstanceValues #1#2#3 {
1170   \xtemplate_show_values:nnn {#1} {#2} {#3}
1171 }

```

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

```

1172 \cs_new_nopar:Npn \IfInstanceExistTF #1#2 {
1173   \xtemplate_if_instance_exist:nnnTF {#1} { } {#2}
1174 }
1175 \cs_new_nopar:Npn \IfInstanceExistT #1#2 {
1176   \xtemplate_if_instance_exist:nnnT {#1} { } {#2}
1177 }
1178 \cs_new_nopar:Npn \IfInstanceExistF #1#2 {
1179   \xtemplate_if_instance_exist:nnnF {#1} { } {#2}
1180 }

```

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

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

```

1181 \cs_new_protected:Npn \EvaluateNow #1 {#1}
1182 \cs_new_protected:Npn \KeyValue #1 {#1}

```

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

```

1183 \cs_new_protected_nopar:Npn \AssignTemplateKeys {
1184   \xtemplate_assignments_pop:
1185 }

```

### 9.3.7 Recent additions to the code

`FP_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  $\langle \textit{dimen} \rangle$  registers that stores the stretch and shrink components. Assignments are global.

```

1186 \cs_new_nopar:Npn \TP_split_finite_skip_value:nnNN #1#2{
1187   \skip_split_finite_else_action:nnNN {#1} {
1188     \PackageError{xtemplate}{Value~ for~ key~ #2~ contains~ ‘fil(11)’}
1189     {Only~ finite~ minus~ or~ plus~ parts~ are~ allowed~ for~ this~ key.}
1190   }
1191 }

1192 </initex | package>

```

## Index

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

Symbols			
\:	..... 267	\c_xtemplate_instances_root_tl	....
\@	..... 267, 268	.. <u>7</u> , 9, 13, 96, 651, 659, 660, 868, 873	
\\	..... 936–938, 943, 947, 953, 954, 960, 961, 967–969, 975, 981, 982, 989, 997, 998, 1004–1006, 1012, 1014, 1020, 1021, 1026, 1031, 1040, 1046, 1052, 1054, 1060, 1064, 1068–1077, 1090, 1096, 1097, 1103	\c_xtemplate_key_order_tl	.....
		..... 13, <u>14</u> , 14, 205, 429, 685	
		\c_xtemplate_keytypes_arg_clist	....
		..... 13, <u>15</u> , 15–18, 308	
		\c_xtemplate_keytypes_root_tl	.....
		.... <u>7</u> , 10, 13, 73, 117, 120, 126, 158	
		\c_xtemplate_restrict_root_tl	.....
		..... <u>7</u> , 11, 13, 137, 162	
		\c_xtemplate_values_root_tl	.....
		..... <u>7</u> , 12, 13, 130, 131, 133, 166	
		\c_xtemplate_vars_root_tl	.....
		..... <u>7</u> , 13, 13, 141, 143, 149, 170	
A		\c_zero	..... 176
\AssignTemplateKeys	..... 4, <u>1183</u> , 1183	\char_make_other:N	..... 268
B		\char_set_lccode:nn	..... 267
\bool_if:cTF	..... 316	\clist_clear:N	..... 427, 593
\bool_if:NF	..... 214, 220, 647	\clist_gset_eq:cN	..... 137
\bool_if:NTF	..... 606, 730, 759, 775, 791, 798, 805, 812	\clist_if_empty:NF	..... 530
\bool_if:nTF	..... 174	\clist_if_empty:NTF	..... 539
\bool_new:N	..... 24–26	\clist_if_in:NnT	..... 253
\bool_set_false:N	.... 272, 578, 645, 713	\clist_if_in:NnTF	..... 550
\bool_set_true:N	. 239, 280, 573, 599, 720	\clist_if_in:NVF	..... 607
C		\clist_map_break:	..... 240, 302
\c_nine	..... 175	\clist_map_function:NN	..... 308, 1061
\c_xtemplate_code_root_tl	..... <u>7</u> , 7, 13, 53, 412, 633, 634, 655, 727, 899	\clist_map_function:nN	..... 218
\c_xtemplate_defaults_root_tl	.....	\clist_map_inline:Nn	..... 531
	..... <u>7</u> , 8, 13, 110, 111, 113, 154	\clist_new:N	..... 15, 27, 37

\clist_put_right:Nn	16–18, 620	\DeclareTemplateCode	4, 997, 1033, 1118, 1124
\clist_remove_element:Nn	551	\DeclareTemplateInterface	3, 1031, 1118, 1121
\clist_set_eq:Nc	161	\dim_gset:Nn	799
\clist_set_eq:NN	516	\dim_new:N	38
\cs_generate_from_arg_count:cNnn	411	\dim_set:Nn	387, 801
\cs_generate_from_arg_count:NNnn	767		
\cs_generate_variant:Nn	65, 87, 311, 349, 355, 625, 682, 711	<b>E</b>	
\cs_gset:Npn	412, 760, 776	\EditCollectionInstance	11, 1118, 1142
\cs_gset_eq:cc	633	\EditInstance	11, 1118, 1139
\cs_if_exist:cTF	53, 61, 72, 96	\EditTemplateDefaults	10, 1118, 1136
\cs_if_exist:NTF	467, 494	\EvaluateNow	11, 89, 1181, 1181
\cs_if_free:cT	110, 130	\exp_after:wN	277, 715, 836
\cs_if_free:cTF	117, 141	\exp_not:c	655, 741
\cs_new:Npn	43, 52, 60, 66, 71, 109, 116, 129, 136, 140, 152, 156, 160, 164, 168, 172, 194, 212, 258, 271, 284, 292, 314, 327, 350, 356, 361, 364, 367, 370, 373, 376, 385, 394, 403, 437, 440, 453, 489, 515, 538, 563, 569, 577, 581, 591, 598, 602, 619, 626, 636, 644, 664, 678, 690, 694, 718, 722, 757, 838, 849, 856, 863, 880, 883, 894, 897, 901, 908, 915, 926, 1064	\exp_not:N	475, 502, 652, 750, 753, 767, 769, 783, 784, 822, 831
\cs_new_eq:NN	42, 818	\exp_not:n	483, 507, 512
\cs_new_nopar:Npn	234, 244, 312, 313, 683, 712, 729, 736, 745, 748, 758, 765, 774, 781, 790, 797, 804, 811, 820, 826, 835, 891, 1172, 1175, 1178, 1186	\exp_not:V	179, 653, 742, 751, 754, 768, 770, 771, 783, 785, 786, 822, 823, 831, 832
\cs_new_protected:Npn	1121, 1124, 1127, 1130, 1133, 1136, 1139, 1142, 1181, 1182	<b>F</b>	
\cs_new_protected_nopar:Npn	1118, 1145, 1148, 1151, 1154, 1157, 1160, 1163, 1166, 1169, 1183	\filedate	3
\cs_set:cpx	651	\filedescription	3
\cs_set:Npn	297, 418, 750, 762, 778	\filename	3
\cs_set_eq:cc	658	\fileversion	3
\cs_set_eq:NN	202, 203, 422, 423, 517, 518, 521, 522, 594, 595	<b>G</b>	
\cs_set_nopar:Npn	295	\g_xtemplate_object_type_prop	13, 19, 19, 44, 67, 181, 190
\cs_show:c	899	\group_begin:	266
		\group_end:	270
<b>D</b>		<b>I</b>	
\DeclareCollectionInstance	9, 1118, 1133	\IfInstanceExistF	1178
\DeclareInstance	7, 1118, 1130	\IfInstanceExistT	1175
\DeclareObjectType	3, 1084, 1118, 1118	\IfInstanceExistTF	10, 1172, 1172
\DeclareRestrictedTemplate	12, 1118, 1127	\int_gset:Nn	792
		\int_new:N	39
		\int_set:Nn	173, 378, 794
		\intexpr_compare:nTF	45
		\intexpr_compare_p:n	175, 176
		\iow_space:	42
		<b>K</b>	
		\KeyValue	11, 81, 838, 1053, 1181, 1182
		\KV_key_no_value_elt:n	202, 422, 517, 521, 594
		\KV_key_value_elt:nn	203, 423, 518, 522, 595







# X

\xtemplate_assign_boolean: .	20, 729, 729
\xtemplate_assign_boolean_aux:n	..... 729, 731, 733, 736
\xtemplate_assign_choice: ..	20, 745, 745
\xtemplate_assign_code: ...	20, 748, 748
\xtemplate_assign_code:n	..... 20, 748, 750, 753, 757
\xtemplate_assign_commalist:	20, 811, 818
\xtemplate_assign_function:	20, 758, 758
\xtemplate_assign_function_aux:N	..... 758, 760, 762, 765
\xtemplate_assign_instance:	20, 774, 774
\xtemplate_assign_instance_aux:N	..... 774, 776, 778, 781
\xtemplate_assign_integer: .	20, 790, 790
\xtemplate_assign_length: ..	20, 790, 797
\xtemplate_assign_skip: ...	20, 790, 804
\xtemplate_assign_tokenlist:	..... 20, 811, 811, 819
\xtemplate_assign_tokenlist_aux:N	..... 811, 813, 815, 820
\xtemplate_assign_variable:N	... 20, 792, 794, 799, 801, 806, 808, 826, 826
\xtemplate_assignments_pop:	..... 21, 891, 891, 1184
\xtemplate_assignments_push:n	..... 21, 652, 894, 894
\xtemplate_convert_to_assignments:	..... 19, 650, 683, 683, 726
\xtemplate_convert_to_assignments_aux:n	..... 683, 688, 690
\xtemplate_convert_to_assignments_aux:nn	..... 683, 694, 711
\xtemplate_convert_to_assignments_aux:nV	..... 683, 692
\xtemplate_declare_instance:nnnnn	..... 19, 636, 636, 1131, 1134
\xtemplate_declare_instance_aux:nnnnn	..... 636, 641, 644, 680
\xtemplate_declare_object_type:nn	..... 16, 172, 172, 1119
\xtemplate_declare_restricted:nnnn	..... 18, 569, 569, 1128
\xtemplate_declare_template_code:nnnnn	..... 18, 403, 403, 1125
\xtemplate_declare_template_keys:nnnn	..... 16, 194, 194, 1122
\xtemplate_edit_defaults:nnn	..... 19, 577, 577, 1137
\xtemplate_edit_defaults_aux:nnn	..... 574, 577, 579, 581
\xtemplate_edit_instance:nnnn	..... 19, 664, 664, 1140, 1143
\xtemplate_edit_instance_aux:nnnnn	..... 664, 678, 682
\xtemplate_edit_instance_aux:nVnnn	..... 664, 670
\xtemplate_execute_if_arg_agree:nnT	..... 14, 43, 43, 197, 406
\xtemplate_execute_if_code_exist:nnT	..... 15, 52, 52, 637, 898, 927
\xtemplate_execute_if_keys_exist:nnT	..... 15, 71
\xtemplate_execute_if_keytype_exist:nT	..... 15, 60, 60, 65
\xtemplate_execute_if_keytype_exist:VF	..... 60
\xtemplate_execute_if_keytype_exist:VT	..... 15, 216
\xtemplate_execute_if_type_exist:nT	..... 15, 66, 66, 195, 404
\xtemplate_find_global:	19, 702, 712, 712
\xtemplate_find_global_aux:w	..... 712, 715, 718
\xtemplate_get_collection:n	..... 21, 864, 883, 883
\xtemplate_if_eval_now:n	..... 88
\xtemplate_if_eval_now:nTF	..... 15, 88, 315, 377, 386, 395
\xtemplate_if_instance_exist:nnn	... 95
\xtemplate_if_instance_exist:nnnF	..... 657, 1179
\xtemplate_if_instance_exist:nnnT	1176
\xtemplate_if_instance_exist:nnnTF	..... 15, 95, 665, 865, 872, 916, 1173
\xtemplate_if_key_value:n	..... 80
\xtemplate_if_key_value:nT	... 15, 80, 87
\xtemplate_if_key_value:VT	..... 15, 80, 737, 827
\xtemplate_if_keys_exist:nnT	..... 71, 408, 570, 582, 902, 909
\xtemplate_if_use_template:n	..... 102
\xtemplate_if_use_template:nTF	..... 16, 102, 850
\xtemplate_implement_choice_elt:n	..... 18, 517, 538, 538, 564
\xtemplate_implement_choice_elt:nn	..... 18, 518, 538, 563

\xtemplate_implement_choices:n	.....	\xtemplate_show_values:nnn	.....
.....	18, 456, 515, 515	.....	21, 901, 915, 1167, 1170
\xtemplate_key_to_value:	.....	\xtemplate_show_vars:nn	21, 901, 926, 1164
.....	20, 738, 828, 835, 835	\xtemplate_space:	.....
\xtemplate_key_to_value_aux:w	.....	.....	42, 42, 329, 334, 358, 524, 566
.....	835, 836, 838	\xtemplate_split_keytype:n	.....
\xtemplate_parse_keys_elt:n	.....	.....	17, 213, 266, 271
.....	17, 202, 212, 212, 259	\xtemplate_split_keytype_arg:n	.....
\xtemplate_parse_keys_elt:nn	.....	.....	17, 289, 292, 292, 311, 699
.....	17, 203, 258, 258	\xtemplate_split_keytype_arg:V	.....
\xtemplate_parse_keys_elt_aux:	.....	.....	17, 292, 340, 446, 543, 555, 622
.....	212, 229, 244	\xtemplate_split_keytype_arg_aux:n	.....
\xtemplate_parse_keys_elt_aux:n	.....	.....	292, 295, 309, 312
.....	212, 219, 234	\xtemplate_split_keytype_arg_aux:w	.....
\xtemplate_parse_values:nn	.....	.....	292, 297, 305, 313
.....	19, 586, 591, 591, 646, 725	\xtemplate_split_keytype_aux:w	.....
\xtemplate_parse_values_elt:n	.....	.....	266, 277, 284
.....	19, 594, 598, 598	\xtemplate_store_defaults:n	.....
\xtemplate_parse_values_elt:nn	.....	.....	16, 109, 109, 207, 425, 587, 628
.....	19, 595, 602, 602	\xtemplate_store_key_implementation:nnn	.....
\xtemplate_parse_values_elt_aux:nn	.....	.....	18, 410, 418, 418
.....	602, 619, 625	\xtemplate_store_keytypes:n	.....
\xtemplate_parse_values_elt_aux:Vn	.....	.....	16, 109, 116, 208, 630
.....	602, 609, 612	\xtemplate_store_restrictions:n	.....
\xtemplate_parse_vars_elt:n	.....	.....	16, 109, 136, 428, 588
.....	18, 422, 437, 437, 521	\xtemplate_store_value_boolean:n	.....
\xtemplate_parse_vars_elt:nn	.....	.....	17, 314, 314
.....	18, 423, 440, 440, 522	\xtemplate_store_value_choice:n	.....
\xtemplate_parse_vars_elt_aux:n	.....	.....	17, 327, 327, 349
.....	447, 453, 453	\xtemplate_store_value_choice:V	.....
\xtemplate_parse_vars_elt_aux:w	.....	.....	17, 327, 528
.....	453, 480, 489	\xtemplate_store_value_choice_aux:n	.....
\xtemplate_recover_defaults:n	16, 152, 152, 419, 584, 627, 639, 723, 904	.....	327, 350, 355
\xtemplate_recover_keytypes:n	.....	\xtemplate_store_value_choice_aux:V	.....
.....	16, 152, 156, 420, 592, 629, 911	.....	327, 331, 336
\xtemplate_recover_restrictions:n	.....	\xtemplate_store_value_choice_name:n	.....
.....	16, 152, 160, 585	.....	17, 261, 327, 356
\xtemplate_recover_values:n	.....	\xtemplate_store_value_code:n	.....
.....	16, 152, 164, 667, 918	.....	17, 361, 361
\xtemplate_recover_vars:n	.....	\xtemplate_store_value_commalist:n	.....
.....	16, 152, 168, 631, 640, 679, 724, 929	.....	17, 361, 364
\xtemplate_set_template_eq:nn	.....	\xtemplate_store_value_function:n	.....
.....	19, 572, 626, 626	.....	17, 361, 367
\xtemplate_show_code:nn	21, 897, 897, 1155	\xtemplate_store_value_instance:n	.....
\xtemplate_show_defaults:nn	.....	.....	17, 361, 370
.....	21, 901, 901, 1158	\xtemplate_store_value_integer:n	.....
\xtemplate_show_keytypes:nn	.....	.....	17, 376, 376
.....	21, 901, 908, 1161	\xtemplate_store_value_length:n	.....
		.....	17, 376, 385

\xtemplate_store_value_skip:n .....	\xtemplate_use_collection:nn .....
..... 17, 376, 394	..... 20, 880, 880, 1152
\xtemplate_store_value_tokenlist:n .	\xtemplate_use_instance:nn .....
..... 17, 361, 373	..... 20, 784, 849, 849, 1149
\xtemplate_store_values:n .....	\xtemplate_use_instance_aux:nn ....
..... 16, 109, 129, 649	..... 849, 853, 863
\xtemplate_store_vars:n .....	\xtemplate_use_instance_aux:nNnnn ..
..... 16, 109, 140, 426, 632	..... 849, 851, 856
\xtemplate_unknown_choice_aux:n ....	\xtemplate_use_template:nnn .....
..... 1061, 1064	..... 20, 722, 722, 858, 1146