The xtemplate package*

The LaTeX3 Project†

2011/01/03

1 Introduction

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

1. Authoring of the text, with mark-up
2. Document layout design
3. Implementation (with TeX programming) of the design

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

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

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

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

*This file has version number 2115, last revised 2011/01/03.

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

- Loading a completely different document class, such as KOMA-Script or memoir, that allows easy customisation.

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

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

2 What is a document?

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

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

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

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

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

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

3 Objects, templates, and instances

By formally declaring our document to be composed of mark-up elements grouped into objects, which are interpreted and typeset with a set of templates, each of which has one or more instances with which to compose each and every semantic unit of the text, we can cleanly separate the components of document construction. The xtemplate package provides the tools to do this.

3.1 Object types

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

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

`\DeclareObjectType` `\DeclareObjectType {`⟨*name*⟩`} {`⟨*Nargs*⟩`}`

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

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

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

3.2 Templates

A *template* is a generalised design solution for representing the information of a specified *object type*. Templates that do the same thing — e.g., two completely different ways of printing a chapter heading — are grouped together by their object type and given separate names. There are two important parts to a template:

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

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

`\DeclareTemplateInterface`

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

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

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

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

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

`\DeclareTemplateCode`

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

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

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

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

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

`\AssignTemplateKeys`

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

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

Key Type	Description of input
`boolean`	`true` or `false`
`choice {`⟨*choices*⟩`}`	A list of pre-defined choices
`code`	Generalised key type; use `#1` as the input to the key
`commalist`	A comma-separated list of arbitrary items
`function` N	A function definition with N arguments (N from 0 to 9)
`instance {`⟨*name*⟩`}`	An instance of type ⟨*name*⟩
`integer`	An integer expression (e.g., `(1 + 5)/2`)
`length`	A dimension expression (e.g., `3pt + 2cm`)
`skip`	A dimension expression with glue (e.g., `3pt plus 2pt minus 1pt`)
`tokenlist`	A 'token list' input; any text or commands

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

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

Table 2: Bindings required for different key types when defining template implementations with `\DeclareTemplateCode`. Starred entries may be prefixed with the keyword `global` to make a global assignment.

3.3 Multiple choices

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

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

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

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

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

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

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

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

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

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

3.4 Instances

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

For example, a template might say 'here is a section with or without a number that might be centred or left aligned and print its contents in a certain font of a certain size, with a bit of a gap before and after it' whereas an instance declares 'this is a section with a number, which is centred and set in 12pt italic with a 10pt skip before and a 12pt skip after it'.

Therefore, an instance is just a frozen version of a template with specific settings as chosen by the designer.

`\DeclareInstance`

`\DeclareInstance {`⟨*object type*⟩`} {`⟨*instance*⟩`} {`⟨*template*⟩`} {`⟨*parameters*⟩`}`
The name of the instance being declared is ⟨*instance*⟩, with ⟨*parameters*⟩ the keyval input to set some or all of the ⟨*template*⟩ keys to specific values.

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

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

3.5 Document interface

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

`\UseInstance`

`\UseInstance {`⟨*object type*⟩`} {`⟨*instance*⟩`}` ⟨*arguments*⟩

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

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

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

`\UseTemplate`

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

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

or use the template directly:

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

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

3.6 Summaries

For the document designer:

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

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

For the class programmer:

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

4 Instances in different contexts

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

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

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

`\DeclareCollectionInstance`

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

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

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

An example might be:

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

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

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

5 Bits 'n' pieces

5.1 Does an instance exist?

`\IfInstanceExist`*TF*

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

5.2 Changing the defaults of a template's keys

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

`\EditTemplateDefaults`

`\EditTemplateDefaults {`⟨*object type*⟩`} {`⟨*template*⟩`} {`⟨*new defaults*⟩`}`

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

5.3 Small changes to an instance

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

`\EditInstance`
`\EditCollectionInstance`

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

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

5.4 Parameters evaluated now

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

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

5.5 Setting one key to the value of another

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

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

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

5.6 When template parameters should be frozen

A class designer may be inheriting templates declared by someone else, either third-party code or the LaTeX kernel itself. Sometimes these templates will be overly general for the

purposes of the document. The user should be able to customise parts of the template instances, but otherwise be restricted to only those parameters allowed by the designer.

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

`\DeclareRestrictedTemplate`

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

6 Getting information about templates and instances

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

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

`\ShowInstanceValues`
`\ShowCollectionInstanceValues`

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

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

7 Examples

(Nothing here yet.)

8 Code documentation

8.1 Variables and constants

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.2 Execute or error functions

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

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

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

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

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

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

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

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

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

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

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

8.3 Utility functions

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

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

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

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

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

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

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

Tests if assignment begins with `\UseTemplate`.

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

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

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

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

8.4 Creating object types

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

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

8.5 Declaring template keys

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

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

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

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

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

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

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

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

8.6 Storing defaults and values

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

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

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

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

8.7 Implementing templates

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

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

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

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

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

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

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

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

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

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

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

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

8.8 Modifying templates

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

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

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

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

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

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

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

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

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

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

8.9 Creating instances

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

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

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

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

`\xtemplate_convert_to_assignments:` `\xtemplate_convert_to_assignments:`

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

`\xtemplate_find_global:` `\xtemplate_find_global:`

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

8.10 Converting values to assignments

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

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

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

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

`\xtemplate_key_to_value:` `\xtemplate_key_to_value:`

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

8.11 Using instances

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

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

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

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

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

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

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

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

`\xtemplate_assignments_pop:` `\xtemplate_assignments_pop:`

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

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

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

8.12 Showing details

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

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

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

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

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

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

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

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

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

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

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

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