The `rkeyval` package: Syntactically restricted key-value scanning

Michael Downes and David M. Jones
American Mathematical Society

Version 2.00, 2004/06/28

Contents

1 Introduction

The `rkeyval` package provides functions for scanning key-value notation similar to the kind of scanning supported by the standard `keyval` package. However, the syntax is more restrictive in order to make some improved error-checking possible. In particular, if a comma is omitted between two instances of `key={value}` form, the `\RestrictedSetKeys` command will spot the missing comma and issue a suitable error message (and it will be given at the point where the missing comma is detected, before reading any further in the TEX file). The standard `\setkeys` command, by contrast, will append the second key name to the value of the first key and discard the second value, without any notification to the user that anything has gone wrong. But that is partly because the standard `\setkeys` command allows implied values and does not require braces around explicit values (except when necessary to hide material that has a syntactic resemblance to a key-value pair). With `\RestrictedSetKeys` the value must always be present and it must be enclosed in braces.

Further restrictions of the `\RestrictedSetKeys` command and its companion commands reduce memory consumption in certain ways. Defining a key creates only one control sequence, a container for holding the value. Processing of key values is normally limited to storing a value given by the user; any additional processing must be supplied separately by the programmer.

Generally speaking, the error-checking done by `\RestrictedSetKeys` is better for applications where all the keys are expected to have textual values, while

`\setkeys` is better when one wants to silently recover as far as possible from syntactic errors, instead of notifying the user of the errors; or when keys have nontrivial default values (i.e., not empty) or other kinds of special processing.

```
\RestrictedSetKeys{setup-code}{group}{code}{key={val}, ...}
```

Normally `\RestrictedSetKeys` simply carries out the following assignment for each key-value pair:

```
\def\group'key{val}
```

The first argument is normally empty, but the exact nature of the warnings given and other aspects of the processing can be affected by putting extra setup code there. The `amsrefs` package uses this to implement a copying operation where field name and value are written out immediately to another file instead of being stored in the usual way.

Some examples for defining the key names associated with a given group. This defines "title" as a recognized key for the `bib` group:

```
\DefineSimpleKey{bib}{title}
```

If a key is defined with `\DefineSimpleKey`, the result of using the same key more than once in a single entry will be an error message.

This defines "title" to be a repeatable key:

```
\DefineSupersedingKey{bib}{title}
```

If it occurs more than once, the last value supersedes the earlier ones, instead of getting an error. This variation is not needed for simple usage, but in more complicated situations where key values are combined from multiple sources, it may be useful.

This defines "author" to be a repeatable key, with each value being appended to a list of values:

```
\DefineAdditiveKey{bib}{author}{\name}
```

The third argument specifies a wrapper function that should be applied to each item in the list. I.e., suppose that two author names are given:

```
author={Smith, J. Q.},
author={Taylor, X. Y.},
```

Then they will be stored in the form

```
\name{Smith, J. Q.}\name{Taylor, X. Y.}
```

This defines "transition" to be a dummy key with a value that is superficially nonempty but effectly empty:

```
\DefineDummyKey{bib}{transition}
```

Defining a dummy key like this can be useful in dealing with certain boundary situations that sometimes arise.

2 Implementation

Standard declaration of package name and date.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{rkeyval}[2004/05/05 v1.08]
```

\@xp
\@nx

```
\let\@xp\expandafter
\let\@nx\noexpand
```

\@gobblethree
\@nilgobble

Not in the LaTeX kernel yet.

```
\long\def\@gobblethree#1#2#3{}
\long\def\@nilgobble#1\@nil{}
```

\@emptytoks

Using `\@ifundefined` here avoids problems with really old versions of LaTeX that choke on `\newtoks` if it is written directly in the false branch of a conditional.

```
\@ifundefined{@emptytoks}{\csname newtoks\endcsname\@emptytoks}{}
```

\@temptokenb

```
\@ifundefined{@temptokenb}{\csname newtoks\endcsname\@temptokenb}{}
```

\@append

```
\def\@append#1#2#3{\@xp\def\@xp#2\@xp{#2#1{#3}}}
```

\star@

Test for a trailing option marked by a star. Usage:

```
\newcommand{\blub}[1]{\star@{\blubaux{#1}}{default}}
```

Arg 1 of `\star@` is the code to be run, arg 2 is the default value of the option (could be empty). If arg 1 is `\moo`, this test discards a star and expands to `\moo` if a star is found, or expands to `\moo{#2}` if not. As the example shows, arg 1 need not be a single token.

```
\def\star@#1#2{%
    \def\star@a##1{#1}%
    \def\star@b{#1{#2}}%
    \futurelet\@let@token\star@test
}

\def\star@test{\ifx*\@let@token \let\star@b\star@a\fi \star@b}
```

Please note: If there is a space before the star, then the star is not treated as an option char.

Why use a star? Since it's already part of standard LaTeX command syntax, it's unlikely to suffer from catcode changes.

Why not just put the star at the beginning in the usual way? It seemed to me that the lack of a trailing option feature was a deficiency in current LaTeX and could be given an experimental implementation in a package like this without any adverse effect on existing documents.

Ensure non-weird catcode for relevant characters.

```
\@ifundefined{NormalCatcodes}{\RequirePackage{pcatcode}\relax}{}
\PushCatcodes\NormalCatcodes
```

`\extract@group` Extracts "group" from `\group'field`.

```
\def\extract@group#1{%
    \@xp\extract@group@a\string#1\@nil
}
```

`\extract@group@a`

```
\def\extract@group@a#1#2'{#2\@nilgobble}
```

3 Data structures

The result of scanning the key/value pairs is an assignment statement for `\rsk@toks`. For example, consider the entry

```
\bib{LelekZ1962}{article}{
    author={Lelek, A.},
    author={Zaremba, D.},
    title={Dimensions of irreducible ...},
    journal={Fund. Math.},
    date={1962/63},
}
```

The scanned result is to assign

```
\global\rsk@toks{%
    \set:bib'author{Lelek, A.}{}%
    \set:bib'author{Zaremba, D.}{}%
    \set:bib'title{Dimensions of irreducible ...}{}%
    \set:bib'journal{Fund. Math.}{}%
    \set:bib'date{1962/63}{}%
}
```

The extra empty arguments on each line are for auxiliary properties (see below). What happens thereafter with `\rsk@toks` depends on the code in the last arg of `\RestrictedSetKeys`.

4 Auxiliary properties

Unfortunately, the previous section isn't the entire story. In addition to the values of each field, we need to store a set of auxiliary properties associated with those values. Note that properties are explicitly associated with *values*, not with keys, because each value of an additive key could have different properties.

All such extra data will be stored in a special field named "aux", with embedded tags to indicate which field each piece of the field is associated with. The extra bits can be extracted on demand using standard techniques, and the primary value of each field is not burdened with any attachments, so that comparisons or scanning of the field contents can remain as simple as possible.

Thus in practice there is at least one bit of auxiliary information in every bib item, and our previous example would have the title language indicated:

```
\DSK@def\bib'title{Eine Bemerkung zur Darstellung von Polynomen
    \"{u}ber Verb\"{a}nden}%
\@append\bib'title\bib'aux{\selectlanguage{german}}%
```

\set@property

```
\def\set@property#1{%
    \begingroup
        \edef\@tempa{\extract@group#1}%
        \edef\@tempa{%
            \@nx\@append\@nx#1\@xp\@nx\csname \@tempa,aux\endcsname
        }%
    \@xp\endgroup
    \@tempa
}
```

\get@property

```
%    \get@property\destination\bib'title
\def\get@property#1#2{%
    \get@nth@property#1#2\m@ne
}
```

\get@nth@property

```
%    \get@nth@property\destination\bib'title N
\def\get@nth@property#1#2#3{%
    \begingroup
        \edef\@tempa{\extract@group#2}%
        \@tempcnta#3\relax
        \@tempcntb\z@
        \@xp\scan@properties\@xp#2\csname \@tempa,aux\endcsname
        \edef\@tempa{\def\@nx#1{\@tempa}}%
    \@xp\endgroup
    \@tempa
}
```

\scan@properties

```
\def\scan@properties#1#2{%
    \begingroup
        \def\@tempa{#1}%
        \let\@tempc\@empty
        \@xp\find@property #2 \@nil\@nil
        \edef\@tempa{\def\@nx\@tempa{\@tempc}}%
    \@xp\endgroup
    \@tempa
}
```

\find@property

```
\def\find@property#1#2{%
```

```
    \ifx\@nil#1%
    \else
        \def\@tempb{#1}%
        \ifx\@tempa\@tempb
            \ifnum\@tempcnta<\z@
                \def\@tempc{#2}%
            \else
                \advance\@tempcntb\@ne
                \ifnum\@tempcntb=\@tempcnta
                    \def\@tempc{#2}%
                \fi
            \fi
        \fi
        \@xp\find@property
    \fi
}
```

`\reset@property`

```
\def\reset@property#1#2{%
    \reset@nth@property#1\m@ne{#2}%
}
```

`\reset@nth@property`

```
%    \reset@nth@property\bib'title N VALUE
\def\reset@nth@property#1#2#3{%
    \begingroup
        \edef\@tempa{\extract@group#1}%
        \@tempcnta#2\relax
        \@temptokena{#3}%
        \toks@\@emptytoks
        \@tempcntb\z@
        \@xp\reset@scan\@xp#1\csname \@tempa,aux\endcsname
        \edef\@tempa{%
            \def\@xp\@nx\csname \@tempa,aux\endcsname{\the\toks@}%
        }%
    \@xp\endgroup
    \@tempa
}
```

`\reset@scan`

```
\def\reset@scan#1#2{%
    \begingroup
        \def\@tempa{#1}%
        \@xp\reset@scan@a #2 \@nil\@nil
        \edef\@tempa{\toks@{\the\toks@}}%
    \@xp\endgroup
    \@tempa
}
```

\find@property

```
\def\reset@scan@a#1#2{%
    \ifx\@nil#1%
    \else
        \def\@tempb{#1}%
        \ifx\@tempa\@tempb
            \ifnum\@tempcnta<\z@
                \@temptokenb\@temptokena
            \else
                \advance\@tempcntb\@ne
                \ifnum\@tempcntb=\@tempcnta
                    \@temptokenb\@temptokena
                \fi
            \fi
        \else
            \@temptokenb{#2}%
        \fi
        \edef\@tempb{%
            \toks@{\the\toks@ \@nx#1{\the\@temptokenb}}%
        }%
        \@tempb
        \@xp\reset@scan@a
    \fi
}
```

5 Some machinery for finite state automata

Coincidentally I needed to write two finite state automaton parsers for two related packages, so after writing them separately I spent some time analyzing the code fragments they shared in common and abstracted them so that the cs names could be shared.

\fsa@l FSA lookahead.

```
\def\fsa@l{\futurelet\@let@token\fsa@t}
```

\fsa@b FSA bypass a token. Don't delete the space at the end!

```
\def\fsa@b{\afterassignment\fsa@l \let\@let@token= }
```

\fsa@c FSA copy a token (not space, bgroup, egroup).

```
\def\fsa@c#1{\aftergroup#1\fsa@l}
```

\fsa@n FSA next action. This is just a placeholder definition.

```
\let\fsa@n\@empty
```

\fsa@t FSA test. This is just a placeholder definition.

```
\let\fsa@t\@empty
```

6 Now some of the real work

`\rsk@toks`

```
\newtoks\rsk@toks
```

`\rkvIfEmpty` Beginning here.

```
\def\rkvIfEmpty#1#2{%
    \@xp\ifx\csname#1'#2\endcsname\@empty
        \@xp\@firstoftwo
    \else
        \@xp\@secondoftwo
    \fi
}
```

`\rkvIfAdditive`

```
\def\rkvIfAdditive#1{%
    \@xp\let\@xp\@let@token \csname \rkv@setter#1\endcsname
    \afterassignment\@nilgobble
    \@xp\let\@xp\@let@token \@let@token \@empty\@empty\@nil
    \ifx\@let@token\DSK@append
        \@xp\@firstoftwo
    \else
        \@xp\@secondoftwo
    \fi
}
```

`\rkv@setter` It irritates me that I can't embed the `\csname` and `\endcsname` in here.

```
\def\rkv@setter#1{set:\@xp\@gobble\string#1}
```

`\rkv@DSAK` Define a simple, superseding, or additive key.

```
\def\rkv@DSAK#1#2{%
    \addto@group@reset#1{\let#1\@empty}%
    \edef\@tempa{\def\csname \rkv@setter#1\endcsname}%
    \@tempa{#2#1}%
}
```

`\rkv@DDK` This function is used for a dummy key whose value (expansion) should be empty but that should appear non-empty to `\rkvIfEmpty`.

```
\def\rkv@DDK#1{%
    \addto@group@reset#1{\def#1{\@empty}}%
    \@xp\let\csname \rkv@setter#1\endcsname\@gobble
}
```

`\DSK@def`

```
\def\DSK@def#1{%
    \ifx#1\@empty\else
        \PackageWarningNoLine{rkeyval}%
            {Key \string#1 should not be repeated}%
    \fi
```

```
    \DSK@redef#1%
}
```

`\DSK@redef` We weed out empty field values for consistency with `\DSK@append`.

```
\def\DSK@redef#1#2{%
    \@ifempty{#2}{\@gobble}{%
        \def#1{#2}%
        \set@property#1
    }%
}
```

`\init@group@reset`

```
\def\init@group@reset#1{%
    \begingroup
        \edef\@tempb{\@xp\@nx\csname #1@reset\endcsname}%
        \@xp\ifx\@tempb\relax
            \@xp\xdef\@tempb{\let \csname #1,aux\endcsname\@nx\@empty}
        \fi
    \endgroup
}
```

`\addto@group@reset`

```
\def\addto@group@reset#1{%
    \begingroup
        \edef\@tempa{\extract@group#1}%
        \init@group@reset\@tempa
        \edef\@tempa{%
            \@nx\g@addto@macro\@xp\@nx\csname\@tempa @reset\endcsname
        }%
    \@xp\endgroup
    \@tempa
}
```

`\DefineSimpleKey`

```
\newcommand{\DefineSimpleKey}[2]{%
    \@xp\rkv@DSAK
        \csname #1'#2\endcsname
        {\DSK@def}%
}
```

`\DefineSupersedingKey`

```
\newcommand{\DefineSupersedingKey}[2]{%
    \@xp\rkv@DSAK
        \csname #1'#2\endcsname
        {\DSK@redef}%
}
```

`\DefineAdditiveKey`

```
\newcommand{\DefineAdditiveKey}[3]{%
```

```
    \@xp\rkv@DSAK
        \csname #1'#2\endcsname
        {\DSK@append#3}%
}
```

`\DSK@append` We weed out empty field values (e.g., `editor={}` or `editor={␣}`) because otherwise an additive field could end up with a value like `\name{}` which appears non-empty to `\rkvIfEmpty` but produces no output on the page.

```
\def\DSK@append#1#2#3{%
    \@ifempty{#3}{\@gobble}{%
        \@append#1#2{#3}%
        \set@property#2
    }%
}
```

`\DefineDummyKey`

```
\newcommand{\DefineDummyKey}[2]{%
    \@xp\rkv@DDK \csname #1'#2\endcsname
}
```

`\RestrictedSetKeys`

```
\newcommand{\RestrictedSetKeys}[3]{%
    \global\rsk@toks\@xp{\csname #2@reset\endcsname}%
    \def\rsk@finish{#3}%
    \gdef\rsk@set{\@xp\rsk@set@a\csname#2'}%
    #1\relax
    \begingroup
        \rsk@changecase
        \aftergroup\rsk@set
```

Start by removing the opening brace.

```
        \let\fsa@t\rsk@z
        \fsa@l
}
```

The aftergroup tokens end up looking like this:

```
\lowercase{\rsk@set FIELDNAME\endcsname}
 --> \@xp\rsk@set@a\csname bib'fieldname\endcsname
 --> \rsk@set@a\bib'abcdef
```

`\rsk@unknown@key`

```
\def\rsk@unknown@key#1{%
    \PackageWarning{rkeyval}{Unknown key: \string#1}%
    \@xp\def\csname\rkv@setter#1\endcsname {\DSK@redef#1}%
}
```

7 The state machine

```
State 0: Skip opening brace (\rsk@z).
  space -> 0
  {     -> 2
  other -> error "Missing open brace"

State 1: Skip comma (\rsk@a).
  space -> 1
  \par  -> 1
  comma -> 2
  @     -> read optional arg; 1
  }     -> 6
  other -> error "Missing comma"; 2

State 2: Find field name (\rsk@b).
  space  -> 2
  \par   -> 2
  comma  -> 2
  letter -> 3
  {      -> error "Missing key name"; 4
  }      -> 6
  other  -> error "Invalid key name character"; 2

State 3: Scan field name (\rsk@c).
  letter -> 3
  comma  -> error "Invalid key name character"; 3
  equal  -> 4
  other punct  -> 3
  space  -> 4
  {      -> error "Missing equal sign"; 4
  }      -> error "Missing equal sign"; 4
  other  -> error "Invalid key name character"; 3

State 4: Skip equals (\rsk@d).
  space  -> 4
  equal  -> 4
  {      -> 5
  other  -> error "Missing { for value of current key"; 5

State 5: Read field value (\rsk@set@a).
  any -> 1

State 6: Done (\rsk@end).
```

`\rsk@z` State 0: Skip opening brace.

```
\def\rsk@z{%
    \ifx\bgroup\@let@token
        \let\fsa@t\rsk@b
        \let\fsa@n\fsa@b
```

```
    \else
        \ifx\@sptoken\@let@token
            \let\fsa@n\fsa@b
        \else
            \rsk@errf
        \fi
    \fi
    \fsa@n
}
```

`\rsk@a` State 1: Skip comma.

```
\def\rsk@a{%
    \ifx\@let@token\@sptoken
        \let\fsa@n\fsa@b
    \else
        \ifx\@let@token\par
            \let\fsa@n\fsa@b
        \else
            \ifx,\@let@token
                \endgroup
                \let\fsa@t\rsk@b
                \let\fsa@n\fsa@b
            \else
                \ifx\egroup\@let@token
                    \endgroup
                    \let\fsa@n\rsk@end
                \else
                    \endgroup
                    \let\fsa@n\rsk@erraa
                \fi
            \fi
        \fi
    \fi
    \fsa@n
}
```

`\rsk@b` State 2: Find field name.

Allow `\par` here to permit a blank line after the end of one key-val pair and the start of the next (perhaps to break up a long list into sections).

```
\def\rsk@b{%
    \ifcat\@nx\@let@token A%
        \let\fsa@t\rsk@c
        \let\fsa@n\fsa@c
    \else
        \ifx\@sptoken\@let@token
            \let\fsa@n\fsa@b
        \else
            \rsk@bb
        \fi
```

```
    \fi
    \fsa@n
}
```

\rsk@bb

```
\def\rsk@bb{%
    \ifx,\@let@token
        \let\fsa@n\fsa@b
    \else
        \ifx\bgroup\@let@token
            \let\fsa@n\rsk@errb
        \else
            \ifx\egroup\@let@token
                \let\fsa@n\rsk@end
            \else
                \ifx\par\@let@token
                    \let\fsa@n\fsa@b
                \else
                    \let\fsa@n\rsk@errc
                \fi
            \fi
        \fi
    \fi
}
```

\rsk@c State 3: Scan field name.

```
\def\rsk@c{%
    \ifcat\@nx\@let@token A%
        \let\fsa@n\fsa@c
    \else
        \ifx\@sptoken\@let@token
            \let\fsa@t\rsk@d
            \let\fsa@n\fsa@b
        \else
            \ifx=\@let@token
                \let\saw@equal T%
                \let\fsa@t\rsk@d
                \let\fsa@n\fsa@b
            \else
                \rsk@cb
            \fi
        \fi
    \fi
    \fsa@n
}
```

\rsk@cb

```
\def\rsk@cb{%
    \ifx,\@let@token
```

```
        \let\fsa@n\rsk@errc
    \else
        \ifcat\@nx\@let@token .%
            \let\fsa@n\fsa@c
        \else
            \ifx\bgroup\@let@token
                \let\fsa@n\rsk@noequal
            \else
                \ifx\egroup\@let@token
                    \let\fsa@n\rsk@noequal
                \else
                    \let\fsa@n\rsk@errc
                \fi
            \fi
        \fi
    \fi
}
```

`\saw@equal`

```
\let\saw@equal=F
```

`\rsk@d` State 4: Skip equals.

If no equal sign ever came along, then give a warning about it and set `\saw@equal` to true so that when `\rsk@noequal` cycles through again it will take the other branch.

```
\def\rsk@d{%
    \ifx\bgroup\@let@token
        \ifx\saw@equal T%
            \aftergroup\endcsname
            \rsk@endcase
            \let\fsa@n\endgroup
        \else
            \let\saw@equal T%
            \let\fsa@n\rsk@noequal
        \fi
    \else
        \ifx\@sptoken\@let@token
            \let\fsa@n\fsa@b
        \else
            \ifx=\@let@token
                \let\saw@equal T%
                \let\fsa@n\fsa@b
            \else
                \let\fsa@n\rsk@erre
            \fi
        \fi
    \fi
    \fsa@n
}
```

`\rsk@casesensitive`

```
\def\rsk@casesensitive{%
    \let\rsk@changecase\@empty
    \let\rsk@endcase\@empty
}
```

`\rsk@startlc`

```
\def\rsk@startlc{\aftergroup\lowercase\aftergroup{\iffalse}\fi}
```

`\rsk@endlc`

```
\def\rsk@endlc{\iffalse{\fi\aftergroup}}
```

`\rsk@lowercase`

```
\def\rsk@lowercase{%
    \let\rsk@changecase\rsk@startlc
    \let\rsk@endcase\rsk@endlc
}
```

```
\rsk@lowercase
```

`\rsk@resume` Here we get improved reporting of error context by changing end-of-line to be different from normal space. If we don't find a comma on the current line, assume it is an error.

```
\def\rsk@resume{%
    \begingroup
        \rsk@changecase
        \aftergroup\rsk@set
        \let\fsa@t\rsk@a
        \begingroup
            \catcode\endlinechar=\active
            \lccode`\~=\endlinechar
            \lowercase{\let~\par}%
            \fsa@l
}
```

`\rsk@set@a` State 5: Read field value.

```
\def\rsk@set@a#1#2{%
    \star@{\rsk@set@b#1{#2}}{}%
}
```

`\rsk@set@b`

```
\def\rsk@set@b#1#2#3{%
    \@xp\ifx \csname\rkv@setter#1\endcsname \relax
        \rsk@unknown@key#1%
    \fi
    \edef\@tempa{\@xp\@nx\csname \rkv@setter#1\endcsname}%
    \toks@\@xp{\@tempa{#2}{#3}}%
    \edef\@tempa{%
        \global\rsk@toks{\the\rsk@toks \the\toks@}%
```

```
    }%
    \@tempa
    \rsk@resume
}
```

`\rsk@end` State 6: Done.

Lets see, now why did I add this?

```
\def\rsk@end{%
        \global\let\rsk@set\rsk@terminate
        \rsk@endcase
    \endgroup
    \endcsname
    \afterassignment\rsk@finish
    \toks@\bgroup
}
```

`\rsk@terminate`

```
\def\rsk@terminate{\@xp\@gobble\csname}
```

`\NoCommaWarning`

```
\def\NoCommaWarning{\PackageWarning{rkeyval}{Missing comma}}%
```

`\NoCommaError`

```
%%  \def\NoCommaError{\rsk@err{Missing comma}\@ehc}
%%
```

`\rsk@nocomma`

```
\def\rsk@nocomma{\NoCommaWarning}
```

`\rsk@err`

```
\def\rsk@err{\PackageError{rkeyval}}
```

`\rsk@errf`

```
\def\rsk@errf{\rsk@err{Missing open brace}\@ehc\rsk@b}
```

`\rsk@erraa`

```
\long\def\rsk@erraa{\rsk@nocomma \let\fsa@t\rsk@b \fsa@l}
```

`\rsk@errb`

```
\def\rsk@errb{\rsk@err{Missing key name}\@ehc\rsk@d}
```

`\rsk@errc`

```
\def\rsk@errc{\rsk@err{Invalid key name character}\@ehc\fsa@b}
```

`\rsk@noequal`

```
\def\rsk@noequal{\rsk@err{Missing equal sign}\@ehc\rsk@d}
```

`\rsk@erre` In this case we guess that the value is without braces but probably terminated with a comma. We want to scan up to the comma in order to get back in synch.

```
\def\rsk@erre#1,{%
        \rsk@err{Missing open brace for key value}\@ehc
        \iffalse{\fi
    \endgroup
    \endcsname
    \rsk@endcase }{#1},%
}
```

```
\PopCatcodes
```

The usual `\endinput` to ensure that random garbage at the end of the file doesn't get copied by `docstrip`.

```
\endinput
```

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.