The Listings Package

Carsten Heinz <cheinz@gmx.de>

2004/09/07 Version 1.3

Abstract

The listings package is a source code printer for LaTeX. You can typeset stand alone files as well as listings with an environment similar to `verbatim` as well as you can print code snippets using a command similar to `\verb`. Many parameters control the output and if your preferred programming language isn't already supported, you can make your own definition.

Preface

Reading this manual If you are experienced with the listings package, you should read the next paragraph "*News and changes*". Otherwise read section *1 Getting started* step by step and go on with section 2.

News and changes This is the third bugfix release. The most noteworthy news for the tortured — I mean experienced — user is a new `\output` routine, which improves the compatibility to other packages. Therefore the temporary key `noxoutput` has been removed.

`texcs` became a family, i.e. you can highlight your own control sequences diffently from the already defined ones: `texcsstyle=[*][`⟨*class number*⟩`]`... and `texcs=[`⟨*classnumber*⟩`]`... Note the added optional star, which also highlights the backslash in front of the control sequence name. And don't forget to check page 28 for a bug.

`keywordstyle` has a small add-on (see page 28) and `extendedchars` is turned on by default.

The following languages were added: Awk (Gnu and Posix), Gnuplot, Handel-C, ByteCode Java (`JVMIS`), Plasm, Promella, and RSL, refer table refuPredefinedLanguages. Moreover, HTML, AspectJ, and MetaPost have been improved. Thanks go to the contributers.

Thanks There are many people I have to thank for fruitful communication, posting their ideas, giving error reports, adding programming languages to `lstdrvrs.dtx`, and so on. Their names are listed in section 3.4.

Trademarks Trademarks appear throughout this documentation without any trademark symbol; they are the property of their respective trademark owner. There is no intention of infringement; the usage is to the benefit of the trademark owner.

User's guide

1 Getting started

1.1 A minimal file

Before using the listings package, you should be familiar with the LaTeX typesetting system. You need not to be an expert. Here is a minimal file for listings.

```
\documentclass{article}
\usepackage{listings}

\begin{document}
\lstset{language=Pascal}

  % Insert Pascal examples here.

\end{document}
```

Now type in this first example and run it through LaTeX.

→ Must I do that really? Yes and no. Some books about programming say this is good. What a mistake! Typing takes time—wasted if the code is clear to you. And if you need that time to understand what is going on, the author of the book should reconsider the concept of presenting the crucial things—you might want to say that about this guide even—or you're simply unexperienced with programming. If only the latter case applies, you should spend more time on reading (good) books about programming, (good) documentations, and (good) source code from other people. Of course you should also make your own experiments. You will learn a lot. However, running the example through LaTeX shows whether the listings package is installed.

→ The example doesn't work. Are the two packages listings and keyval installed on your system? Consult the administration tool of your TeX distribution, your system administrator, the local TeX and LaTeX guides, a TeX FAQ, and section 4.18—in the given order. If you've checked *all* these sources and are still helpless, you might want to write a post to a TeX newsgroup like `comp.text.tex`.

→ Should I read the software license before using the package? Yes, but read this *Getting started* section first to decide whether you are willing to use the package.

1.2 Typesetting listings

Three types of source codes are supported: code snippets, code segments, and listings of stand alone files; snippets are placed inside paragraphs and the others as separate paragraphs—the difference is the same as between text style and display style formulas.

→ No matter what kind of source you have, if a listing contains national characters like é, Ł, ä, or whatever, you must tell it the package! Section *2.5 Special characters* discusses this issue.

Code snippets The well-known LaTeX command `\verb` typesets code snippets verbatim. The new command `\lstinline` pretty-prints the code, for example '**var** i:**integer**;' is typeset by '`\lstinline!var i:integer;!`'. The exclamation marks delimit the code and can be replaced by any character not in the code; `\lstinline$var i:integer;$` gives the same result.

Displayed code The `lstlisting` environment typesets the enclosed source code. Like most examples, the following one shows verbatim LaTeX code on the right and the result on the left. You might take the right-hand side, put it into the minimal file, and run it through LaTeX.

```
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case insensitive ');
WritE('Pascal keywords.');
```

```
\begin{lstlisting}
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case insensitive ');
WritE('Pascal keywords.');
\end{lstlisting}
```

It can't be easier.

→ That's not true. The name '`listing`' is shorter. Indeed. But other packages already define environments with that name. To be compatible with such packages, all commands and environments of the listings package use the prefix '`lst`'.

The environment provides an optional argument. It tells the package to perform special tasks, for example, to print only the lines 2–5:

```
begin
    { do nothing }
end;
```

```
\begin{lstlisting}[firstline=2,
                   lastline=5]
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case insensitive ');
WritE('Pascal keywords.');
\end{lstlisting}
```

→ Hold on! Where comes the frame from and what is it good for? You can put frames around all listings except code snippets. You will learn it later. The frame shows that empty lines at the end of listings aren't printed. This is line 5 in the example.

→ Hey, you can't drop my empty lines! You can tell the package not to drop them: The key '`showlines`' controls these empty lines and is described in section 4.2. Warning: First read ahead on how to use keys in general.

→ I get obscure error messages when using '`firstline`'. That shouldn't happen. Make a bug report as described in section *6 Troubleshooting*.

Stand alone files Finally we come to `\lstinputlisting`, the command used to pretty-print stand alone files. It has one optional and one file name argument. Note that you possibly need to specify the relative path to the file. Here now the result is printed below the verbatim code since both together don't fit the text width.

```
\lstinputlisting[lastline=4]{listings.sty}
```

%%
%% This is file 'listings.sty',
%% generated with the docstrip utility.
%%

→ The spacing is different in this example. Yes. The two previous examples have aligned columns, i.e. columns with identical numbers have the same horizontal position—this package makes small adjustments only. The columns in the example here are not aligned. This is explained in section 2.10 (keyword: full flexible column format).

Now you know all pretty-printing commands and environments. It remains to learn the parameters which control the work of the listings package. This is, however, the main task. Here are some of them.

1.3 Figure out the appearance

Keywords are typeset bold, comments in italic shape, and spaces in strings appear as ␣. You don't like these settings? Look at this:

```
\lstset{% general command to set parameter(s)
    basicstyle=\small,          % print whole listing small
    keywordstyle=\color{black}\bfseries\underbar,
                                % underlined bold black keywords
    identifierstyle=,           % nothing happens
    commentstyle=\color{white}, % white comments
    stringstyle=\ttfamily,      % typewriter type for strings
    showstringspaces=false}     % no special string spaces
```

```
for i:=maxint to 0 do
begin

end;

Write('Case insensitive ');
WritE('Pascal keywords.');
```

```
\begin{lstlisting}
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case insensitive ');
WritE('Pascal keywords.');
\end{lstlisting}
```

→ You've requested white coloured comments, but I can see the comment on the left side. There are a couple of possible reasons: (1) You've printed the documentation on nonwhite paper. (2) If you are viewing this documentation as a .dvi-file, your viewer seems to have problems with colour specials. Try to print the page on white paper. (3) If a printout on white paper shows the comment, the colour specials aren't suitable for your printer or printer driver. Recreate the documentation and try it again—and ensure that the color package is well-configured.

The styles use two different kinds of commands. `\ttfamily` and `\bfseries` both take no arguments but `\underbar` does; it underlines the following argument. In general, the *very last* command might read exactly one argument, namely some material the package typesets. There's one exception. The last command of `basicstyle` *must not* read any tokens—or you will get deep in trouble.

→ '`basicstyle=\small`' looks fine, but comments look really bad with '`commentstyle=\tiny`' and empty basic style, say. Don't use different font sizes in a single listing.

→ But I really want it! No, you don't.

Warning You should be very careful with striking styles; the recent example is rather moderate—it can get horrible. *Always use decent highlighting.* Unfortunately it is difficult to give more recommendations since they depend on the type of document you're creating. Slides or other presentations often require more striking styles than books, for example. In the end, it's *you* who have to find the golden mean!

Listing 1: A floating example

```
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case␣insensitive␣');
WritE('Pascal␣keywords.');
```

1.4 Seduce to use

You know all pretty-printing commands and some main parameters. Here now comes a small and incomplete overview of other features. The table of contents and the index also provide information.

Line numbers are available for all displayed listings, e.g. tiny numbers on the left, each second line, with 5pt distance to the listing:

```
\lstset{numbers=left, numberstyle=\tiny, stepnumber=2, numbersep=5pt}
```

```
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case␣insensitive␣');
WritE('Pascal␣keywords.');
```

```
\begin{lstlisting}
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case insensitive ');
WritE('Pascal keywords.');
\end{lstlisting}
```

- → I can't get rid of line numbers in subsequent listings. 'numbers=none' turns them off.
- → Can I use these keys in the optional arguments? Of course. Note that optional arguments modify values for one particular listing only: you change the appearance, step or distance of line numbers for a single listing. The previous values are restored afterwards.

The environment allows you to interrupt your listings: you can end a listing and continue it later with the correct line number even if there are other listings in between. Read section 2.6 for a thorough discussion.

Floating listings Displayed listings may float:

```
\begin{lstlisting}[float,caption=A floating example]
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case insensitive ');
WritE('Pascal keywords.');
\end{lstlisting}
```

Don't care about the parameter `caption` now. And if you put the example into the minimal file and run it through LaTeX, please don't wonder: you'll miss the horizontal rules since they are described elsewhere.

→ LaTeX's float mechanism allows to determine the placement of floats. What's about that? You can write '`float=tp`', for example.

Other features There are still features not mentioned so far: automatic breaking of long lines, the possibility to use LaTeX code in listings, automated indexing, or personal language definitions. One more little teaser? Here you are. But note that the result is not produced by the LaTeX code on the right alone. The main parameter is hidden.

if (i$\leq$0) **then** i $\leftarrow$ 1;
if (i$\geq$0) **then** i $\leftarrow$ 0;
if (i$\neq$0) **then** i $\leftarrow$ 0;

```
\begin{lstlisting}
if (i<=0) then i := 1;
if (i>=0) then i := 0;
if (i<>0) then i := 0;
\end{lstlisting}
```

You're not sure whether you should use listings? Read the next section!

1.5 Alternatives

→ Why do you list alternatives? Well, it's always good to know the competitors.

→ I've read the descriptions below and the listings package seems to incorporate all the features. Why should I use one of the other programs? Firstly, the descriptions give a taste and not a complete overview, secondly, listings lacks some properties, and, eventually, you should use the program matching your needs most precisely.

This package is certainly not the final utility for typesetting source code. Other programs do their job very well—if you are not satisfied with listings. Some are independent of LaTeX, other come as separate program plus LaTeX package, and other more are packages which don't pretty-print the source code. The second type inlcudes converters, cross compilers, and preprocessors. Such programs create LaTeX files you can use in your document or stand alone ready-to-run LaTeX files.

Note that I'm not dealing with any literate programming tool here, which could also be an alternative. However, you should have heard of the `WEB` system, the tool Prof. Donald E. Knuth developed and made use of to document and implement TeX.

a2ps started as 'ASCII to PostScript' converter, but today you can invoke the program with `--pretty-print=`⟨*language*⟩ option. If your favourite programming language is not already supported, you can write your own so-called style sheet. You can request line numbers, borders, headers, multiple pages per sheet, and many more. You can even print symbols like $\forall$ or α instead of their verbose forms. If you just want program listings and not a document with some listings, this is the best choice.

LGrind is a cross compiler and comes with many predefined programming languages. For example, you can put the code on the right in your document, invoke LGrind with `-e` option (and file names), and run the created file through LaTeX. You should get a result similar to the left-hand side:

for i:=maxint **to** 0 **do**
begin
{ *do nothing* }
end;

Write(`'Case insensitive '`);
WritE(`'Pascal keywords.'`);

```
%[
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case insensitive ');
WritE('Pascal keywords.');
%]
```

If you use `%(` and `%)` instead of `%[` and `%]`, you get a code snippet instead of a displayed listing. Moreover you can get line numbers to the left or right, use arbitrary LaTeX code in the source code, print symbols instead of verbose names, make font setup, and more. You will (have to) like it (if you don't like listings).

Note that LGrind contains code with a no-sell license and is thus nonfree software.

cvt2ltx is a family of 'source code to LaTeX' converters for C, Objective C, C++, IDL and Perl. Different styles, line numbers and other qualifiers can be chosen by command-line option. Unfortunately it isn't documented how other programming languages can be added.

C++2LaTeX is a C/C++ to LaTeX converter. You can specify the fonts for comments, directives, keywords, and strings, or the size of a tabulator. But as far as I know you can't number lines.

SLaTeX is a pretty-printing Scheme program (invokes LaTeX automatically) especially designed for Scheme and other Lisp dialects. It supports stand alone files, text and display listings, and you can even nest the commands/environments if you use LaTeX code in comments, for example. Keywords, constants, variables, and symbols are definable and use of different styles is possible. No line numbers.

tiny_c2ltx is a C/C++/Java to LaTeX converter based on cvt2ltx (or the other way round?). It supports line numbers, block comments, LaTeX code in/as comments, and smart line breaking. Font selection and tabulators are hard-coded, i.e. you have to rebuild the program if you want to change the appearance.

listing —note the missing s—is not a pretty-printer and the aphorism about documentation at the end of `listing.sty` is not true. It defines `\listoflistings` and a nonfloating environment for listings. All font selection and indention must be done by hand. However, it's useful if you have another tool doing that work, e.g. LGrind.

alg provides essentially the same functionality as algorithms. So read the next paragraph and note that the syntax will be different.

algorithms goes a quite different way. You describe an algorithm and the package formats it, for example

if $i \leq 0$ **then**
  $i \leftarrow 1$
else
  if $i \geq 0$ **then**
    $i \leftarrow 0$
  end if
end if

```
\begin{algorithmic}
\IF{$i\leq0$}
\STATE $i\gets1$
\ELSE\IF{$i\geq0$}
\STATE $i\gets0$
\ENDIF\ENDIF
\end{algorithmic}
```

As this example shows, you get a good looking algorithm even from a bad looking input. The package provides a lot more constructs like `for`-loops, `while`-loops, or comments. You can request line numbers, 'ruled', 'boxed' and floating algorithms, a list of algorithms, and you can customize the terms **if**, **then**, and so on.

pretprin is a package for pretty-printing texts in formal languages—as the title in TUGboat, Volume 19 (1998), No. 3 states. It provides environments which pretty-print *and* format the source code. Analyzers for Pascal and Prolog are defined; adding other languages is easy—if you are or get a bit familiar with automatons and formal languages.

alltt defines an environment similar to `verbatim` except that `\`, `{` and `}` have their usual meanings. This means that you can use commands in the verbatims, e.g. select different fonts or enter math mode.

moreverb requires verbatim and provides verbatim output to a file, 'boxed' verbatims and line numbers.

verbatim defines an improved version of the standard `verbatim` environment and a command to input files verbatim.

fancyvrb is, roughly spoken, a super set of alltt, moreverb, and verbatim, but many more parameters control the output. The package provides frames, line numbers on the left or on the right, automatic line breaking (difficult), and more. For example, an interface to listings exists, i.e. you can pretty-print source code automatically. The package fvrb-ex builds above fancyvrb and defines environments to present examples similar to the ones in this guide.

2 The next steps

Now, before actually using the listings package, you should *really* read the software license. It does not cost much time and provides information you probably need to know.

2.1 Software license

The files `listings.dtx` and `listings.ins` and all files generated from only these two files are referred to as 'the listings package' or simply 'the package'. `lstdrvrs.dtx` and the files generated from that file are 'drivers'.

Contacts Read section *6 Troubleshooting* on how to submit a bug report. Send all other comments, ideas, and additional programming languages to `cheinz@gmx.de` using `listings` as part of the subject.

2.2 Package loading

As usual in LaTeX, the package is loaded by `\usepackage[`⟨*options*⟩`]{listings}`, where `[`⟨*options*⟩`]` is optional and gives a comma separated list of options. Each either loads an additional listings aspect, or changes default properties. Usually you don't have to take care of such options. But in some cases it could be necessary: if you want to compile documents created with an earlier version of this package or if you use special features. Here's an incomplete list of possible options.

→ Where is a list of all options? In the developer's guide since they were introduced to debug the package more easily. Read section 7 on how to get that guide.

`0.21`

compiles a document created with version 0.21.

`draft`

The package prints no stand alone files, but shows the captions and defines the corresponding labels. Note that a global `\documentclass`-option `draft` is recognized, so you don't need to repeat it as a package option.

`final`

Overwrites a global `draft` option.

`savemem`

tries to save some of TeX's memory. If you switch between languages often, it could also reduce compile time. But all this depends on the particular document and its listings.

Note that various experimental features also need explicit loading via options. Read the respective lines in section 5.

After package loading it is recommend to load all used dialects of programming languages with the following command. It is faster to load several languages with one command than loading each language on demand.

`\lstloadlanguages{`⟨*comma separated list of languages*⟩`}`

Each language is of the form `[`⟨*dialect*⟩`]`⟨*language*⟩. Without the optional `[`⟨*dialect*⟩`]` the package loads a default dialect. So write '`[Visual]C++`' if you want Visual C++ and '`[ISO]C++`' for ISO C++. Both together can be loaded by the command `\lstloadlanguages{[Visual]C++,[ISO]C++}`.

Table 1 on page 12 shows all defined languages and their dialects.

2.3 The key=value interface

This package uses the keyval package from the graphics bundle by David Carlisle. Each parameter is controlled by an associated key and a user supplied value. For example, `firstline` is a key and `2` a valid value for this key.

The command `\lstset` gets a comma separated list of "key=value" pairs. The first list with more than a single entry is on page 4: `firstline=2,lastline=5`.

→ So I can write '`\lstset{firstline=2,lastline=5}`' once for all? No. '`firstline`' and '`lastline`' belong to a small set of keys which are used on individual listings. However, your command is not illegal—it has no effect. You have to use these keys inside the optional argument of the environment or input command.

→ What's about a better example of a key=value list? There is one in section 1.3.

→ '`language=[77]Fortran`' does not work inside an optional argument. You must put braces around the value if a value with optional argument is used inside an optional argument. In the case here write '`language={[77]Fortran}`' to select Fortran 77.

→ If I use the '`language`' key inside an optional argument, the language isn't active when I typeset the next listing. All parameters set via '`\lstset`' keep their values up to the end of the current environment or group. Afterwards the previous values are restored. The optional parameters of the two pretty-printing commands and the '`lstlisting`' environment take effect on the particular listing only, i.e. values are restored immediately. For example, you can select a main language and change it for special listings.

→ `\lstinline` has an optional argument? Yes. And from this fact comes a limitation: you can't use the left bracket '`[`' as delimiter except you specify at least an empty optional argument as in '`\lstinline[][var i:integer;[`'. If you forget this, you will either get a "runaway argument" error from TEX, or an error message from the keyval package.

2.4 Programming languages

You already know how to activate programming languages—at least Pascal. An optional parameter selects particular dialects of a language. For example, `language=[77]Fortran` selects Fortran 77 and `language=[XSC]Pascal` does the same for Pascal XSC. The general form is `language=[`⟨*dialect*⟩`]`⟨*language*⟩. If you want to get rid of keyword, comment, and string detection, use `language={}` as argument to `\lstset` or as optional argument.

Table 1 shows all predefined languages and dialects. Use the listed names as ⟨*language*⟩ and ⟨*dialect*⟩, respectively. If no dialect or 'empty' is given in the table, just don't specify a dialect. Each underlined dialect is default; it is selected if you leave out the optional argument. The predefined defaults are the newest language versions or standard dialects.

→ How can I define default dialects? Check section 4.5 for '`defaultdialect`'.

→ I have C code mixed with assembler lines. Can listings pretty-print such source code, i.e. highlight keywords and comments of both languages? '`alsolanguage=[`⟨*dialect*⟩`]`⟨*language*⟩' selects a language additionally to the active one. So you only have to write a language definition for your assembler dialect, which doesn't interfere with the definition of C, say. Moreover you might want to use the key '`classoffset`' described in section 4.5.

→ How can I define my own language? This is discussed in section 4.17. And if you think that other people could benefit by your definition, you might want to send it to the address in section 2.1. Then it will be published under the LATEX Project Public License.

Note that the arguments ⟨*language*⟩ and ⟨*dialect*⟩ are case insensitive and that spaces have no effect.

2.5 Special characters

Tabulators You might get unexpected output if your sources contain tabulators. The package assumes tabulator stops at columns 9, 17, 25, 33, and so on. This is predefined via `tabsize=8`. If you change the eight to the number n, you will get tabulator stops at columns $n+1, 2n+1, 3n+1$, and so on.

```
123456789
  { one tabulator }
    { two tabs }
123     { 123 + two tabs }
```

```
\lstset{tabsize=2}
\begin{lstlisting}
123456789
    { one tabulator }
        { two tabs }
123     { 123 + two tabs }
\end{lstlisting}
```

Table 1: Predefined languages. Note that some definitions are preliminary, for example HTML and XML. Each underlined dialect is default dialect

ABAP (`R/2 4.3`, `R/2 5.0`, `R/3 3.1`, `R/3 4.6C`, <u>`R/3 6.10`</u>)	
ACSL	Ada (`83`, <u>`95`</u>)
Algol (`60`, <u>`68`</u>)	Ant
Assembler (`x86masm`)	Awk (<u>`gnu`</u>, `POSIX`)
bash	Basic (`Visual`)
C (<u>`ANSI`</u>, `Handel`, `Objective`, `Sharp`)	
C++ (`ANSI`, `GNU`, <u>`ISO`</u>, `Visual`)	Caml (<u>`light`</u>, `Objective`)
Clean	Cobol (`1974`, <u>`1985`</u>, `ibm`)
Comal 80	csh
Delphi	Eiffel
Elan	erlang
Euphoria	Fortran (`77`, `90`, <u>`95`</u>)
GCL	Gnuplot
Haskell	HTML
IDL (empty, `CORBA`)	inform
Java (empty, `AspectJ`)	JVMIS
ksh	Lisp (empty, `Auto`)
Logo	make (empty, `gnu`)
Mathematica (`1.0`, <u>`3.0`</u>)	Matlab
Mercury	MetaPost
Miranda	Mizar
ML	Modula-2
MuPAD	NASTRAN
Oberon-2	OCL (`decorative`, <u>`OMG`</u>)
Octave	Oz
Pascal (`Borland6`, <u>`Standard`</u>, `XSC`)	Perl
PHP	PL/I
Plasm	POV
Prolog	Promela
Python	R
Reduce	Rexx
RSL	Ruby
S (empty, `PLUS`)	SAS
Scilab	sh
SHELXL	Simula (<u>`67`</u>, `CII`, `DEC`, `IBM`)
SQL	tcl (empty, `tk`)
TeX (`AlLaTeX`, `common`, `LaTeX`, <u>`plain`</u>, `primitive`)	
VBScript	Verilog
VHDL (empty, `AMS`)	VRML (<u>`97`</u>)
XML	XSLT

For better illustration, the left-hand side uses `tabsize=2` but the verbatim code `tabsize=4`. Note that `\lstset` modifies the values for all following listings in the same environment or group. This is no problem here since the examples are typeset inside minipages. If you want to change settings for a single listing, use the optional argument.

Visible tabulators and spaces One can make spaces and tabulators visible:

```
␣␣␣␣for␣i:=maxint␣to␣0␣do
␣␣␣␣begin
——————→{␣do␣nothing␣}
␣␣␣␣end;
```

```
\lstset{showspaces=true,
        showtabs=true,
        tab=\rightarrowfill}
\begin{lstlisting}
    for i:=maxint to 0 do
    begin
    { do nothing }
    end;
\end{lstlisting}
```

If you request `showspaces` but no `showtabs`, tabulators are converted to visible spaces. The default definition of `tab` produces a 'wide visible space' ␣. So you might want to use `$\to$`, `$\dashv$` or something else instead.

→ Some sort of advice: (1) You should really indent lines of source code to make listings more readable. (2) Don't indent some lines with spaces and others via tabulators. Changing the tabulator size (of your editor or pretty-printing tool) completely disturbs the columns. (3) As a consequence, never share your files with differently tab sized people!

→ To make the LaTeX code more readable, I indent the environments' program listings. How can I remove that indention in the output? Read 'How to gobble characters' in section 7.

Form feeds Another special character is a form feed causing an empty line by default. `formfeed=\newpage` would result in a new page every form feed. Please note that such definitions (even the default) might get in conflict with frames.

National characters If you type in such characters directly as characters of codes 128–255 and use them also in listings, let the package know it—or you'll get really funny results. `extendedchars=true` allows and `extendedchars=false` prohibits extended characters in listings. If you use them, you should load fontenc, inputenc and/or any other package which defines the characters.

→ I have problems using inputenc together with listings. This could be a compatibility problem. Make a bug report as described in section *6 Troubleshooting*.

The extended characters don't cover Arabic, Chinese, Hebrew, Japanese, and so on. Read section 7 for details on work-arounds.

2.6 Line numbers

You already know the keys `numbers`, `numberstyle`, `stepnumber`, and `numbersep` from section 1.4. Here now we deal with continued listings. You have two options to get consistent line numbering across listings.

```
100 for i:=maxint to 0 do
    begin
102     { do nothing }
    end;
```

And we continue the listing:

```
    Write('Case␣insensitive␣');
105 WritE('Pascal␣keywords.');
```

```
\begin{lstlisting}[firstnumber=100]
for i:=maxint to 0 do
begin
    { do nothing }
end;

\end{lstlisting}
And we continue the listing:
\begin{lstlisting}[firstnumber=last]
Write('Case insensitive ');
WritE('Pascal keywords.');
\end{lstlisting}
```

In the example, `firstnumber` is initially set to 100; some lines later the value is `last`, which continues the numbering of the last listing. Note that the empty line at the end of the first part is not printed here, but it counts for line numbering. You should also notice that you can write `\lstset{firstnumber=last}` once and get consecutively numbered code lines—except you specify something different for a particular listing.

On the other hand you can use `firstnumber=auto` and name your listings. Listings with identical names (case sensitive!) share a line counter.

```
  for i:=maxint to 0 do
2 begin
      { do nothing }
4 end;
```

And we continue the listing:

```
6 Write('Case␣insensitive␣');
  WritE('Pascal␣keywords.');
```

```
\begin{lstlisting}[name=Test]
for i:=maxint to 0 do
begin
    { do nothing }
end;

\end{lstlisting}
And we continue the listing:
\begin{lstlisting}[name=Test]
Write('Case insensitive ');
WritE('Pascal keywords.');
\end{lstlisting}
```

The next `Test` listing goes on with line number 8, no matter whether there are other listings in between.

→ Okay. And how can I get decreasing line numbers? Sorry, what? Decreasing line numbers as on page 31. May I suggest to demonstrate your individuality by other means? If you differ, you should try a negative '`stepnumber`' (together with '`firstnumber`').

Read section 7 on how to reference line numbers.

2.7 Layout elements

It's always a good idea to structure the layout by vertical space, horizontal lines, or different type sizes and typefaces. The best to stress whole listings are—not all at once—colours, frames, vertical space, and captions. The latter are also good to refer to listings, of course.

Vertical space The keys `aboveskip` and `belowskip` control the vertical space above and below displayed listings. Both keys get a dimension or skip as value and are initialized to `\medskipamount`.

Frames The key `frame` takes the verbose values `none`, `leftline`, `topline`, `bottomline`, `lines` (top and bottom), `single` for single frames, or `shadowbox`.

```
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

```
\begin{lstlisting}[frame=single]
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

→ The rules aren't aligned. This could be a bug of this package or a problem with your `.dvi` driver. *Before* sending a bug report to the package author, modify the parameters described in section 4.11 heavily. And do this step by step! For example, begin with '`framerule=10mm`'. If the rules are misaligned by the same (small) amount as before, the problem does not come from the rule width. So continue with the next parameter.

Alternatively you can control the rules at the `t`op, `r`ight, `b`ottom, and `l`eft directly by using the four initial letters for single rules and their upper case versions for double rules.

```
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

```
\begin{lstlisting}[frame=trBL]
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

Note that a corner is drawn if and only if both adjacent rules are requested. You might think that the lines should be drawn up to the edge, but what's about round corners? The key `frameround` must get exactly four characters as value. The first character is attached to the upper right corner and it continues clockwise. '`t`' as character makes the corresponding corner round.

```
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

```
\lstset{frameround=fttt}
\begin{lstlisting}[frame=trBL]
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

Note that `frameround` has been used together with `\lstset` and thus the value affects all following listings in the same group or environment. Since the listing is inside a `minipage` here, this is no problem.

→ Dont' use frames all the time, in particular not with short listings. This would emphasize nothing. Use frames for 10% or even less of your listings, for your most important ones.

→ If you use frames on floating listings, do you really want frames? No, I want to separate floats from text. Then it is better to redefine LaTeX's '`\topfigrule`' and '`\botfigrule`'. For example, you could write '`\renewcommand*\topfigrule{\hrule\kern-0.4pt\relax}`' and make the same definition for `\botfigrule`.

Captions Now we come to `caption` and `label`. You might guess that they can be used in the same manner as LaTeX's `\caption` and `\label` commands:

```
\begin{lstlisting}[caption={Useless code},label=useless]
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

Listing 2: Useless code

```
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

Afterwards you could refer to the listing via `\ref{useless}`. By default such a listing gets an entry in the list of listings, which can be printed with the command `\lstlistoflistings`. The key `nolol` suppresses an entry for both the environment or the input command. Moreover, you can specify a short caption for the list of listings: `caption={[`⟨*short*⟩`]`⟨*long*⟩`}`. Note that the whole value is enclosed in braces since an optional value is used in an optional argument.

If you don't want the label `Listing` plus number, you should use `title`:

```
\begin{lstlisting}[title={‘Caption’ without label}]
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

‘Caption’ without label

```
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

→ Something goes wrong with ‘`title`’ in my document: in front of the title is a delimiter. The result depends on the document class; some are not compatible. Contact the package author for a work-around.

Colours One more element. You need the color package and can then request coloured background via `backgroundcolor=`⟨*color command*⟩.

→ Great! I love colours. Fine, yes, really. And I like to remind you of the warning about striking styles on page 5.

```
\lstset{backgroundcolor=\color{yellow}}
```

```
for i:=maxint to 0 do
begin
    j:=square(root(i));
end;
```

```
\begin{lstlisting}[frame=single,
                   framerule=0pt]
for i:=maxint to 0 do
begin
    j:=square(root(i));
end;
\end{lstlisting}
```

The example also shows how to get coloured space around the whole listing: use a frame whose rules has no width.

2.8 Emphasize identifiers

Recall the pretty-printing commands and environment. `\lstinline` prints code snippets, `\lstinputlisting` whole files, and `lstlisting` pieces of code which reside in the LaTeX file. And what are these different 'types' of source code good for? Well, it just happens that a sentence contains a code fragment. Whole files are typically included in or as an appendix. Nevertheless some books about programming also include such listings in normal text sections—to increase the number of pages. Nowadays source code should be shipped on disk or CD-ROM and only the main header or interface files should be typeset for reference. So, please, don't misuse the listings package. But let's get back to the topic.

Obviously '`lstlisting` source code' isn't used to make an executable program from. Such source code has some kind of educational purpose or even didactic.

→ What's the difference between educational and didactic? Something educational can be good or bad, true or false. Didactic is true by definition.

Usually *keywords* are highlighted when the package typesets a piece of source code. This isn't necessary for readers knowing the programming language well. The main matter is the presentation of interface, library or other functions or variables. If this is your concern, here come the right keys. Let's say, you want to emphasize the functions `square` and `root`, for example, by underlining them. Then you could do it like this:

```
\lstset{emph={square,root},emphstyle=\underbar}
```

```
for i:=maxint to 0 do
begin
    j:=square(root(i));
end;
```

```
\begin{lstlisting}
for i:=maxint to 0 do
begin
    j:=square(root(i));
end;
\end{lstlisting}
```

→ Note that the list of identifiers `{square,root}` is enclosed in braces. Otherwise the keyval package would complain about an undefined key `root` since the comma finishes the key=value pair. Note also that you *must* put braces around the value if you use an optional argument of a key inside an optional argument of a pretty-printing command. Though it is not necessary, the following example uses these braces. They are typically forgotten when they become necessary,

Both keys have an optional ⟨*class number*⟩ argument for multiple identifier lists:

```
\lstset{emph={square},       emphstyle=\color{red},
        emph={[2]root,base},emphstyle={[2]\color{blue}}}
```

```
for i:=maxint to 0 do
begin
    j:=square(root(i));
end;
```

```
\begin{lstlisting}
for i:=maxint to 0 do
begin
    j:=square(root(i));
end;
\end{lstlisting}
```

→ What is the maximal ⟨*class number*⟩? $2^{31} - 1 = 2\,147\,483\,647$. But TeX's memory will exceed before you can define so many different classes.

One final hint: Keep the lists of identifiers disjoint. Never use a keyword in an 'emphasize' list or one name in two different lists. Even if your source code is highlighted as expected, there is no guarantee that it is still the case if you change the order of your listings or if you use the next release of this package.

2.9 Indexing

is just like emphasizing identifiers—I mean the usage:

```
\lstset{index={square},index={[2]root}}
```

```
for i:=maxint to 0 do
begin
    j:=square(root(i));
end;
```

```
\begin{lstlisting}
for i:=maxint to 0 do
begin
    j:=square(root(i));
end;
\end{lstlisting}
```

Of course, you can't see anything here. You will have to look at the index.

→ Why the '`index`' key is able to work with multiple identifier lists? This question is strongly related to the '`indexstyle`' key. Someone might want to create multiple indexes or want to insert prefixes like '`constants`', '`functions`', '`keywords`', and so on. The '`indexstyle`' key works like the other style keys except that the last token *must* take an argument, namely the (printable form of the) current identifier.
You can define '`\newcommand\indexkeywords[1]{\index{keywords, #1}}`' and make similar definitions for constant or function names. Then '`indexstyle=[1]\indexkeywords`' might meet your purpose. This becomes easier if you want to create multiple indexes with the index package. If you have defined appropriate new indexes, it is possible to write '`indexstyle=\index[keywords]`', for example.

→ Let's say, I want to index all keywords. It would be annoying to type in all the keywords again, specifically if the used programming language changes frequently. Just read ahead.

The `index` key has in fact two optional arguments. The first is the well-known ⟨*class number*⟩, the second is a comma separated list of other keyword classes whose identifiers are indexed. The indexed identifiers then change automatically with the defined keywords—not automagically, it's not an illusion.

Eventually you need to know the names of the keyword classes. It's usually the key name followed by a class number, for example, `emph2`, `emph3`, ..., `keywords2` or `index5`. But there is no number for the first order classes `keywords`, `emph`, `directives`, and so on.

→ '`index=[keywords]`' does not work. The package can't guess which optional argument you mean. Hence you must specify both if you want to use the second one. You should try '`index=[1][keywords]`'.

2.10 Fixed and flexible columns

The first thing a reader notices—except different styles for keywords, etc.—is the column alignment. Arne John Glenstrup invented the flexible column format in 1997. Since then some efforts were made to develop this branch farther. Currently three column formats are provided: fixed, flexible, and full flexible. Take a close look at the following examples.

`columns=`	`fixed` (at 0.6em)	`flexible` (at 0.45em)	`fullflexible` (at 0.45em)

`WOMEN   are`
`        MEN`
`WOMEN are`
`better MEN`
	WOMEN are
	MEN
	WOMEN are
	better MEN
		WOMEN are
		MEN
		WOMEN are
		better MEN
			WOMEN are
			MEN
			WOMEN are
			better MEN

→ Why are women better men?   Do you want to philosophize? Well, have I ever said that the statement "women are better men" is true? I can't even remember this about "women are men" . . .

In the abstract one can say: The fixed column format ruins the spacing intended by the font designer, while the flexible formats ruin the column alignment (possibly) intended by the programmer. Common to all is that the input characters are translated into a sequence of basic output units like

i f		x	=	y		**t h e n**		**w r i t e**	(	'	a l i g n	'	)
						e l s e		p r i n t	(	'	a l i g n	'	) ;

Now, the fixed format puts n characters into a box of width $n \times$ 'base width', where the base width is 0.6em in the example. The format shrinks and stretches the space between the characters to make them fit the box. As shown in the example, some character strings look b a d or worse, but the output is vertically aligned.

If you don't need or like this, you should use a flexible format. All characters are typeset at their natural width. In particular, they never overlap. If a word requires more space than reserved, the rest of the line simply moves to the right. The difference between the two formats is that the full flexible format cares about nothing else and the normal flexible format tries to fix the column alignment if a character string needs less space than 'reserved'. In the flexible example above, the two MENs are vertically aligned since some space has been inserted in the fourth line to fix the alignment. In the full flexible format, the two MENs are not aligned.

Note that both flexible modes printed the two blanks in the first line as a single blank, but for different reasons: the normal flexible format fixes the column alignment and the full flexible format doesn't care about the second space.

3 Advanced techniques

3.1 Style definitions

It is obvious that a pretty-printing tool like this requires some kind of language selection and definition. The first has already been described and the latter is convered by the next section. However, it is very convenient to have the same for printing styles: at a central place of your document they can be modified easily and the changes take effect on all listings.

Similar to languages, `style=`⟨*style name*⟩ activates a previously defined style. A definition is as easy: `\lstdefinestyle{`⟨*style name*⟩`}{`⟨*key=value list*⟩`}`. Keys not used in such a definition are untouched by the corresponding style selection, of course. For example, you could write

```
\lstdefinestyle{numbers}
    {numbers=left, stepnumber=1, numberstyle=\tiny, numbersep=10pt}
\lstdefinestyle{nonumbers}
    {numbers=none}
```

and switch from listings with line numbers to listings without ones and vice versa simply by `style=nonumbers` and `style=numbers`, respectively.

→ You could even write '\lstdefinestyle{C++}{language=C++,style=numbers}'. Style and language names are independent of each other and so might coincide. Moreover it is possible to activate other styles.

→ It's easy to crash the package using styles. Write '\lstdefinestyle{crash}{style=crash}' and '\lstset{style=crash}'. TEX's capacity will exceed, sorry [parameter stack size]. Only bad boys use such recursive calls, but only good girls use this package. Thus the problem is of minor interest.

3.2 Language definitions

This is like style definitions except for an optional dialect name and an optional base language—and, of course, a different command name and specialized keys. In the simple case it's `\lstdefinelanguage{`⟨*language name*⟩`}{`⟨*key=value list*⟩`}`. For many programming languages it is sufficient to specify keywords and standard function names, comments, and strings. Let's look at an example.

```
\lstdefinelanguage{rock}
  {morekeywords={one,two,three,four,five,six,seven,eight,
      nine,ten,eleven,twelve,o,clock,rock,around,the,tonight},
   sensitive=false,
   morecomment=[l]{//},
   morecomment=[s]{/*}{*/},
   morestring=[b]",
  }
```

There isn't much to say about keywords. They are defined like identifiers you want to emphasize. Additionally you need to specify whether they are case sensitive or not. And yes: you could insert `[2]` in front of the keyword `one` to define the keywords as 'second order' and print them in `keywordstyle={[2]...}`.

→ I get a '`Missing = inserted for \ifnum`' error when I select my language. Did you forget the comma after '`keywords={...}`'? And if you encounter unexpected characters after selecting a language (or style), you have probably forgotten a different comma or you have given to many arguments to a key, for example, `morecomment=[l]{--}{!}`.

So let's turn to comments and strings. Each value starts with a *mandatory* `[`⟨*type*⟩`]` argument followed by a changing number of opening and closing delimiters. Note that each delimiter (pair) requires a key=value on its own, even if types are equal. Hence, you'll need to insert `morestring=[b]'` if single quotes open and close string or character literals in the same way as double quotes do in the example.

Eventually you need to know the types and their numbers of delimiters. The reference guide contains full lists, here we discuss only the most common. For strings these are `b` and `d` with one delimiter each. This delimiter opens and closes the string and inside a string it is either escaped by a backslash or it is doubled. The comment type `l` requires exactly one delimiter, which starts a comment on any column. This comment goes up to the end of line. The other two most common comment types are `s` and `n` with two delimiters each. The first delimiter opens a comment which is terminated by the second delimiter. In contrast to the `s`-type, `n`-type comments can be nested.

```
\lstset{morecomment=[l]{//},
        morecomment=[s]{/*}{*/},
        morecomment=[n]{(*}{*)},
        morestring=[b]",
        morestring=[d]'}
```

```
"str\"ing "     not a string
'str''ing '     not a string
// comment line
/* comment/**/ not a comment
(* nested (**) still comment
   comment  *) not a comment
```

```
\begin{lstlisting}
"str\"ing "     not a string
'str''ing '     not a string
// comment line
/* comment/**/ not a comment
(* nested (**) still comment
   comment  *) not a comment
\end{lstlisting}
```

→ Is it *that* easy? Almost. There are some troubles you can run into. For example, if '-*' starts a comment line and '-*-' a string (unlikely but possible), then you must define the shorter delimiter first. Another problem: by default some characters are not allowed inside keywords, for example '-', ':', '.', and so on. The reference guide covers this problem by introducing some more keys, which let you adjust the standard character table appropriately. But note that white space characters are prohibited inside keywords.

Finally remember that this section is only an introduction to language definitions. There are more keys and possibilities.

3.3 Delimiters

You already know two special delimiter classes: comments and strings. However, their full syntax hasn't been described so far. For example, `commentstyle` applies to all comments—except you specify something different. The *optional* `[`⟨*style*⟩`]` argument follows the *mandatory* `[`⟨*type*⟩`]` argument.

```
\lstset{morecomment=[l][keywordstyle]{//},
        morecomment=[s][\color{white}]{/*}{*/}}
```

```
// bold comment line
a single
```

```
\begin{lstlisting}
// bold comment line
a single /* comment */
\end{lstlisting}
```

As you can see, you have the choice between specifying the style explicitly by LaTeX commands or implicitly by other style keys. But, you're right, some implicitly defined styles have no seperate keys, for example the second order keyword style. Here—and never with the number 1—you just append the order to the base key: `keywordstyle2`.

You ask for an application? Here you are: one can define different printing styles for 'subtypes' of a comment, for example

```
\lstset{morecomment=[s][\color{blue}]{/*+}{*/},
        morecomment=[s][\color{red}]{/*-}{*/}}
```

```
/*  normal comment */
/*+    keep cool   */
/*-     danger!    */
```

```
\begin{lstlisting}
/*  normal comment */
/*+    keep cool   */
/*-     danger!    */
\end{lstlisting}
```

Here, the comment style is not applied to the second and third line.

→ Please remember that both 'extra' comments must be defined *after* the normal comment, since the delimiter '/*' is a substring of '/*+' and '/*-'.

→ I have another question. Is 'language=⟨*different language*⟩' the only way to remove such additional delimiters? Call `deletecomment` and/or `deletestring` with the same arguments to remove the delimiters (but you don't need to provide the optional style argument).

Eventually, you might want to use the prefix `i` on any comment type. Then the comment is not only invisible, it is completely discarded from the output!

```
\lstset{morecomment=[is]{/*}{*/}}
```

begin **end**
beginend
	`\begin{lstlisting}`
	`begin /* comment */ end`
	`begin/* comment */end`
	`\end{lstlisting}`

Okay, and now for the real challenges. More general delimiters can be defined by the key `moredelim`. Legal types are `l` and `s`. These types can be preceded by an `i`, but this time *only the delimiters* are discarded from the output. This way you can select styles by markers.

```
\lstset{moredelim=[is][\ttfamily]{|}{|}}
```

roman `typewriter`	`\begin{lstlisting}`
	`roman |typewriter|`
	`\end{lstlisting}`

You can even let the package detect keywords, comments, strings, and other delimiters inside the contents.

```
\lstset{moredelim=*[s][\itshape]{/*}{*/}}
```

/ **begin***
(comment *)*
*'␣string␣' */*
	`\begin{lstlisting}`
	`/* begin`
	`  (* comment *)`
	`   ' string ' */`
	`\end{lstlisting}`

Moreover, you can force the styles being applied cumulative.

```
\lstset{moredelim=**[is][\ttfamily]{|}{|}, % cumulative
        moredelim=*[s][\itshape]{/*}{*/}}  % not so
```

/ **begin***
'␣string␣'
*`typewriter` */*
 	
`begin`
`'␣string␣'`
`/*typewriter*/`
	`\begin{lstlisting}`
	`/* begin`
	`   ' string '`
	`   |typewriter| */`
	
	`| begin`
	` ' string '`
	` /*typewriter*/ |`
	`\end{lstlisting}`

Look carefully at the output and note the differences. The second `begin` is not printed in bold typewriter type since standard LaTeX has no such font.

This suffices for an introduction. Now go and find some more applications.

3.4 Closing and credits

You've seen a lot of keys but you are far away from knowing all of them. The next step is the real use of the listings package. Please take the following advices. Firstly, look up the known commands and keys in the reference guide to get a notion of the notation there. Secondly, poke about around these keys to learn some other parameters. Then, hopefully, you'll be prepared if you encounter any problems or need some special things.

→ There is one question 'you' haven't asked all the last pages: who is to blame. The author has written the guides, coded the listings package and some language drivers. Other people defined more languages or contributed their ideas; many others made bug reports, but only the first bug finder is listed. Special thanks go to (alphabetical order)

> Andreas Bartelt, Jan Braun, Denis Girou, Arne John Glenstrup, Frank Mittelbach, Rolf Niepraschk, Rui Oliveira, Jens Schwarzer, and Boris Veytsman.

Moreover I wish to thank

> Bjørn Ådlandsvik, Gaurav Aggarwal, Jason Alexander, Holger Arndt, Donald Arseneau, David Aspinall, Claus Atzenbeck, Michael Bachmann, Luca Balzerani, Peter Bartke (big thankyou), Heiko Bauke, Oliver Baum, Ralph Becket, Andres Becerra Sandoval, Kai Below, Matthias Bethke, Javier Bezos, Olaf Trygve Berglihn, Geraint Paul Bevan, Peter Biechele, Beat Birkhofer, Frédéric Boulanger, Martin Brodbeck, Walter E. Brown, Achim D. Brucker, Ján Buša, David Carlisle, Bradford Chamberlain, Brian Christensen, Neil Conway, Patrick Cousot, Xavier Crégut, Christopher Creutzig, Holger Danielsson, Andreas Deininger, Robert Denham, Detlev Dröge, Anders Edenbrandt, Mark van Eijk, Norbert Eisinger, Brian Elmegaard, Jon Ericson, Thomas Esser, Chris Edwards, David John Evans, Tanguy Fautré, Ulrike Fischer, Robert Frank, Martine Gautier Daniel Gazard, Daniel Gerigk, Dr. Christoph Giess, KP Gores, Adam Grabowski, Jean-Philippe Grivet, Christian Gudrian, Jonathan de Halleux, Carsten Hamm, Martina Hansel, Harald Harders, Christian Haul, Aidan Philip Heerdegen, Jim Hefferon, Heiko Heil, Jürgen Heim, Alvaro Herrera, Richard Hoefter, Dr. Jobst Hoffmann, Torben Hoffmann, Morten Høgholm, Berthold Höllmann, Gérard Huet, Hermann Hüttler, Ralf Imhäuser, R. Isernhagen, Oldrich Jedlicka, Dirk Jesko, Loïc Joly, Christian Kaiser, Bekir Karaoglu, Marcin Kasperski, Christian Kindinger, Steffen Klupsch, Peter Köller (big thankyou), Reinhard Kotucha, Stefan Lagotzki, Tino Langer, Rene H. Larsen, Olivier Lecarme, Thomas Leduc, Dr. Peter Leibner, Thomas Leonhardt (big thankyou), Magnus Lewis-Smith, Knut Lickert, Dan Luecking, Peter Löffler, Kris Luyten, José Romildo Malaquias, Andreas Matthias, Patrick TJ McPhee, Brooks Moses, Riccardo Murri, Knut Müller, Svend Tollak Munkejord, Gerd Neugebauer, Torsten Neuer, Michael Niedermair, Xavier Noria, Heiko Oberdiek, Xavier Olive, Alessio Pace, Markus Pahlow, Morten H Pedersen, Zvezdan V. Petkovic, Michael Piefel, Michael Piotrowski, Manfred Piringer, Vincent Poirriez, Adam Prugel-Bennett, Ralf Quast, Aslak Raanes, Venkatesh Prasad Ranganath, Jeffrey Ratcliffe, Georg Rehm, Fermin Reig, Detlef Reimers, Stephen Reindl, Franz Rinnerthaler, Peter Ruckdeschel, Magne Rudshaug, Jonathan Sauer, Vespe Savikko, Gunther Schmidl, Andreas Schmidt, Walter Schmidt, Jochen Schneider, Benjamin Schubert, Uwe Siart, Axel Sommerfeldt, Richard Stallman, Nigel Stanger, Martin Steffen, Andreas Stephan, Stefan Stoll, Enrico Straube, Werner Struckmann, Martin Süßkraut, Gabriel Tauro, Winfried Theis, Jens T. Berger Thielemann, William Thimbleby, Arnaud Tisserand, Jens Troeger, Kalle Tuulos, Gregory Van Vooren, Timothy Van Zandt, Jörg Viermann, Thorsten Vitt, Herbert Voss (big thankyou), Edsko de Vries, Herfried Karl Wagner, Dominique de Waleffe, Bernhard Walle, Jared Warren, Michael Weber, Sonja Weidmann, Andreas Weidner, Herbert Weinhandl, Robert Wenner, Michael Wiese, James Willans, Jörn Wilms, Kai Wollenweber, Ulrich G. Wortmann, Cameron H.G. Wright, and Andrew Zabolotny.

There are probably other people who contributed to this package. If I've missed your name, send an email.

Reference guide

4 Main reference

Your first training is completed. Now that you've left the User's guide, the friend telling you what to do has gone. Get more practice and become a journeyman!

→ Actually, the friend hasn't gone. There are still some advices, but only from time to time.

4.1 How to read the reference

Commands, keys and environments are presented as follows.

hints `command`, `environment` or `key` with ⟨*parameters*⟩ `default`

This field contains the explanation; here we describe the other fields.

If present, the label in the left margin provides extra information: '*addon*' indicates additionally introduced functionality, '*changed*' a modified key, '*data*' a command just containing data (which is therefore adjustable via `\renewcommand`), and so on. Some keys and functionality are '*bug*'-marked or with a †-sign. These features might change in future or could be removed, so use them with care.

If there is verbatim text touching the right margin, it is the predefined value. Note that some keys default to this value every listing, namely the keys which can be used on individual listings only.

The label in the right margin is the current version number and marks newly introduced features.

Regarding the parameters, please keep in mind the following:

1. A list always means a comma separated list. You must put braces around such a list. Otherwise you'll get in trouble with the keyval package; it complains about an undefined key.

2. You must put parameter braces around the whole value of a key if you use an `[`⟨*optional argument*⟩`]` of a key inside an optional `[`⟨*key=value list*⟩`]`: `\begin{lstlisting}[caption={[one]two}]`.

3. Brackets '`[ ]`' usually enclose optional arguments and must be typed in verbatim. Normal brackets '[ ]' always indicate an optional argument and must not be typed in. Thus `[*]` must be typed in exactly as is, but [*] just gets `*` if you use this argument.

4. A vertical rule indicates an alternative, e.g. ⟨`true`|`false`⟩ allows either `true` or `false` as arguments.

5. If you want to enter one of the special characters `{}#%\`, this character must be escaped with a backslash. This means that you must write `\}` for the single character 'right brace'—but of course not for the closing paramater character.

4.2 Typesetting listings

`\lstset{`⟨*key=value list*⟩`}`

sets the values of the specified keys, see also section 2.3. The parameters keep their values up to the end of the current group. In opposition, all optional ⟨*key=value list*⟩s below modify the parameters for single listings only.

`\lstinline[`⟨*key=value list*⟩`]`⟨*character*⟩⟨*source code*⟩⟨*same character*⟩

works like `\verb` but respects the active language and style. These listings use flexible columns except requested differently in the optional argument. You can write '`\lstinline!var i:integer;!`' and get '**var** i:**integer**;'.

Since the command first looks ahead for an optional argument, you must provide at least an empty one if you want to use `[` as ⟨*character*⟩.

† An experimental implementation has been done to support the syntax `\lstinline[`⟨*key=value list*⟩`]{`⟨*source code*⟩`}`. Try it if you want and report success and failure. A known limitation is that inside another argument the last source code token must not be an explicit space token—and, of course, using a listing inside another argument is itself experimental, see section 5.1.

`\begin{lstlisting}[`⟨*key=value list*⟩`]`

`\end{lstlisting}`

typesets the code in between as a displayed listing.

In contrast to the environment of the verbatim package, LaTeX code on the same line and after the end of environment is typeset respectively executed.

`\lstinputlisting[`⟨*key=value list*⟩`]{`⟨*file name*⟩`}`

typesets the stand alone source code file as a displayed listing.

4.3 Space and placement

`float=[*]`⟨*subset of* `tbph`⟩ or `float` `floatplacement`

makes sense on individual displayed listings only and lets them float. The argument controls where LaTeX is *allowed* to put the float: at the top or bottom of the current/next page, on a separate page, or here where the listing is.

The optional star can be used to get a double-column float in a two-column document.

`floatplacement=`⟨*place specifiers*⟩ `tbp`

is used as place specifier if `float` is used without value.

`aboveskip=`⟨*dimension*⟩ `\medskipamount`

`belowskip=`⟨*dimension*⟩ `\medskipamount`

define the space above and below displayed listings.

† `lineskip=`⟨*dimension*⟩ `0pt`

specifies additional space between lines in listings.

† boxpos=⟨b|c|t⟩ c

Sometimes the listings package puts a \hbox around a listing—or it couldn't be printed or even processed correctly. The key determines the vertical alignment to the surrounding material: bottom baseline, centered or top baseline.

4.4 The printed range

print=⟨true|false⟩ or print true

controls whether an individual displayed listing is typeset. Even if set false, the respective caption is printed and the label is defined.

Note: If the package is loaded without the draft option, you can use this key together with \lstset. In the other case the key can only be used to typeset particular listings despite of the draft option.

firstline=⟨*number*⟩ 1

lastline=⟨*number*⟩ 9999999

can be used on individual listings only. They determine the physical input lines used to print displayed listings.

linerange={⟨*first1*⟩-⟨*last1*⟩,⟨*first2*⟩-⟨*last2*⟩, and so on}

can be used on individual listings only. The given line ranges of the listing are displayed. The intervals must be sorted and must not intersect.

showlines=⟨true|false⟩ or showlines false

If true, the package prints empty lines at the end of listings. Otherwise these lines are dropped (but they count for line numbering).

emptylines=[*]⟨*number*⟩

sets the maximum of empty lines allowed. If there is a block of more than ⟨*number*⟩ empty lines, only ⟨*number*⟩ ones are printed. Without the optional star, line numbers can be disturb when blank lines are omitted; with the star, the lines keep their original numbers.

gobble=⟨*number*⟩ 0

gobbles ⟨*number*⟩ characters at the beginning of each *environment* code line. This key has no effect on \lstinline or \lstinputlisting.

Tabulators expand to tabsize spaces before they are gobbled. Code lines with less than gobble characters are considered empty, but never indent the end of environment by more characters.

4.5 Languages and styles

Please note that the arguments ⟨*language*⟩, ⟨*dialect*⟩, and ⟨*style name*⟩ are case insensitive and that spaces have no effect.

style=⟨*style name*⟩ {}

activates the key=value list stored with \lstdefinestyle.

`\lstdefinestyle{`⟨*style name*⟩`}{`⟨*key=value list*⟩`}`

stores the key=value list.

`language=[`⟨*dialect*⟩`]`⟨*language*⟩ `{}`

activates a (dialect of a) programming language. The 'empty' default language detects no keywords, no comments, no strings, and so on. Without specifying ⟨*dialect*⟩, the package chooses a default dialect.

Table 1 on page 12 lists all languages and dialects provided by `lstdrvrs.dtx`. The predefined default dialects are underlined.

`alsolanguage=[`⟨*dialect*⟩`]`⟨*language*⟩

selects the (dialect of a) programming language additionally to the current active one. Note that some language definitions interfere with each other and are plainly incompatible.

Take a look at the `classoffset` key in section 4.6 if you want to highlight the keywords of the languages differently.

`defaultdialect=[`⟨*dialect*⟩`]`⟨*language*⟩

defines ⟨*dialect*⟩ as default dialect for ⟨*language*⟩. If you have defined a default dialect other than empty, for example `defaultdialect=[iama]fool`, you can't select the empty dialect, even not with `language=[]fool`.

Eventually here's a small list of language specific keys.

optional `printpod=`⟨`true`|`false`⟩ `false`

prints or drops PODs in Perl.

renamed,optional `usekeywordsintag=`⟨`true`|`false`⟩ `true`

The package either use the first order keywords in tags or prints all identifiers inside `<>` in keyword style.

optional `tagstyle=`⟨*style*⟩ `{}`

determines the style in which tags and their content is printed.

optional `markfirstintag=`⟨*style*⟩ `false`

prints the first name in tags with keyword style.

optional `makemacrouse=`⟨`true`|`false`⟩ `true`

Make specific: Macro use of identifiers, which are defined as first order keywords, also prints the surrounding `$(` and `)` in keyword style. e.g. you could get **$(strip** $(BIBS)**)**. If deactivated you get $(**strip** $(BIBS)).

4.6 Figure out the appearance

`basicstyle=`⟨*basic style*⟩ `{}`

is selected at the beginning of each listing. You could use `\footnotesize`, `\small`, `\itshape`, `\ttfamily`, or something like that. The last token of ⟨*basic style*⟩ must not read any following characters.

`identifierstyle=`⟨*style*⟩ `{}`

commentstyle=⟨*style*⟩ \itshape

stringstyle=⟨*style*⟩ {}

determine the style for non-keywords, comments, and strings. The *last* token might be an one-parameter command like \textbf or \underbar.

addon keywordstyle=[⟨*number*⟩][*]⟨*style*⟩ \bfseries

deprecated ndkeywordstyle=⟨*style*⟩ keywordstyle

are used to print keywords and second order keywords (if defined). The optional ⟨*number*⟩ argument is the class number to which the style should be applied. ndkeywordstyle has the fixed class number 2.

Add-on: If you use the optional star after the (optional) class number, the keywords are printed uppercase — even if a language is case sensitive and defines lowercase keywords only. Maybe there should also be an option for lowercase keywords . . .

classoffset=⟨*number*⟩ 0

is added to all class numbers before the styles, keywords, identifiers, etc. are assigned. The example below defines the keywords directly; you could do it indirectly by selection two different languages.

```
\lstset{classoffset=0,
        morekeywords={one,three,five},keywordstyle=\color{red},
        classoffset=1,
        morekeywords={two,four,six},keywordstyle=\color{blue},
        classoffset=0}% restore default
```

```
one two three
four five six
```

```
\begin{lstlisting}
one two three
four five six
\end{lstlisting}
```

addon,bug,optional texcsstyle=[*][⟨*class number*⟩]⟨*style*⟩ keywordstyle

optional directivestyle=⟨*style*⟩ keywordstyle

determine the style of TEX control sequences and directives. Note that these key are present only if you've chosen an appropriate language.

The optional star of texcsstyle also highlights the backslash in front of the control sequence name. Note that this option is set for all texcs lists.

Bug: texcs... interferes with other keyword lists. If, for example, emph contains the word foo, then the control sequence \foo will show up in emphstyle.

emph=[⟨*number*⟩]{⟨*identifier list*⟩}

moreemph=[⟨*number*⟩]{⟨*identifier list*⟩}

deleteemph=[⟨*number*⟩]{⟨*identifier list*⟩}

emphstyle=[⟨*number*⟩]{⟨*style*⟩}

define, add and remove the ⟨*identifier list*⟩ from 'emphasize class ⟨*number*⟩' respectively define the style for that class. If you don't give an optional argument, the package assumes ⟨*number*⟩ $= 1$.

These keys are described more detailed in section 2.8.

delim=[*[*]][⟨*type*⟩][[⟨*style*⟩]]⟨*delimiter(s)*⟩

moredelim=[*[*]][⟨*type*⟩][[⟨*style*⟩]]⟨*delimiter(s)*⟩

deletedelim=[*[*]][⟨*type*⟩]⟨*delimiter(s)*⟩

deletes all previously defined delimiters (but neither strings nor comments) and defines the user supplied delimiter, adds the specified delimiter, or removes it.

In the first two cases ⟨*style*⟩ is used to print the delimited code (and the delimiters). Here, ⟨*style*⟩ could be something like `\bfseries` or `\itshape`, or it could refer to other styles via `keywordstyle`, `keywordstyle2`, `emphstyle`, etc.

Supported types are `l` and `s`, see the comment keys in section 3.2 for an explanation. If you use the prefix `i`, i.e. `il` or `is`, the delimiters are not printed, which is some kind of invisibility.

If you use one optional star, the package will detect keywords, comments, and strings inside the delimited code. With both optional stars, aditionally the style is applied cumulative, see section 3.3.

4.7 Getting all characters right

extendedchars=⟨`true`|`false`⟩ or extendedchars `true`

allows or prohibits extended characters in listings, that means (national) characters of codes 128–255. If you use extended characters, you should load fontenc and/or inputenc, for example.

inputencoding=⟨*encoding*⟩ `{}`

determines the input encoding. The usage of this key requires the inputenc package; nothing happens if it's not loaded.

upquote=⟨`true`|`false`⟩ `false`

determines whether the left and right quote are printed ‘’ or `'. This key requires the textcomp package.

tabsize=⟨*number*⟩ `8`

sets tabulator stops at columns ⟨*number*⟩+1, 2·⟨*number*⟩+1, 3·⟨*number*⟩+1, and so on. Each tabulator in a listing moves the current column to the next tabulator stop.

showtabs=⟨`true`|`false`⟩ `false`

make tabulators visible or invisible. A visible tabulator looks like ⎯⎯⎯, but that can be changed. If you choose invisible tabulators but visible spaces, tabulators are converted to an appropriate number of spaces.

tab=⟨*tokens*⟩

⟨*tokens*⟩ is used to print a visible tabulator. You might want to use `$\to$`, `$\mapsto$`, `$\dashv$` or something like that instead of the strange default definition.

showspaces=⟨`true`|`false`⟩ `false`

lets all blank spaces appear ␣ or as blank spaces.

showstringspaces=⟨`true`|`false`⟩ `true`

lets blank spaces in strings appear ␣ or as blank spaces.

formfeed=⟨*tokens*⟩ `\bigbreak`

Whenever a listing contains a form feed ⟨*tokens*⟩ is executed.

4.8 Line numbers

numbers=⟨`none`|`left`|`right`⟩ `none`

makes the package either print no line numbers, or put them on the left or the right side of a listing.

stepnumber=⟨*number*⟩ `1`

All lines with "line number $\equiv 0$ modulo ⟨*number*⟩" get a line number. If you turn line numbers on and off with `numbers`, the parameter `stepnumber` will keep its value. Alternatively you can turn them off via `stepnumber=0` and on with a nonzero number and keep the value of `numbers`.

numberfirstline=⟨`true`|`false`⟩ `false`

The first line of each listing gets numbered (if numbers are on at all) even if the line number is not divisible by `stepnumber`.

numberstyle=⟨*style*⟩ `{}`

determines the font and size of the numbers.

numbersep=⟨*dimension*⟩ `10pt`

is the distance between number and listing.

numberblanklines=⟨`true`|`false`⟩ `true`

If this is set to false, blank lines get no printed line number.

firstnumber=⟨`auto`|`last`|⟨*number*⟩⟩ `auto`

`auto` lets the package choose the first number: a new listing starts with number one, a named listing continues the most recent same-named listing (see below), and a stand alone file begins with the number corresponding to the first input line.

`last` continues the numbering of the most recent listing and ⟨*number*⟩ sets it to the number.

name=⟨*name*⟩

names a listing. Displayed environment-listings with the same name share a line counter.

data `\thelstnumber` `\arabic{lstnumber}`

prints the lines' numbers.

We show an example on how to redefine `\thelstnumber`. But if you test it, you won't get the result shown on the left.

```
\renewcommand*\thelstnumber{\oldstylenums{\the\value{lstnumber}}}
```

```
753  begin { empty lines }
752
751
750
749
748
747
746  end; { empty lines }
```

```
\begin{lstlisting}[numbers=left,
                   firstnumber=753]
begin { empty lines }






end; { empty lines }
\end{lstlisting}
```

→ The example shows a sequence $n, n+1, \ldots, n+7$ of 8 three-digit figures such that the sequence contains each digit $0, 1, \ldots, 9$. But 8 is not minimal with that property. Find the minimal number and prove that it is minimal. How many minimal sequences do exist?
Now look at the generalized problem: Let $k \in \{1, \ldots, 10\}$ be given. Find the minimal number $m \in \{1, \ldots, 10\}$ such that there is a sequence $n, n+1, \ldots, n+m-1$ of m k-digit figures which contains each digit $\{0, \ldots, 9\}$. Prove that the number is minimal. How many minimal sequences do exist?
If you solve this problem with a computer, write a TeX program!

4.9 Captions

In despite of LaTeX standard behaviour, captions and floats are independent from each other here; you can use captions with non-floating listings.

`title=`⟨*title text*⟩

is used for a title without any numbering or label.

`caption={[`⟨*short*⟩`]`⟨*caption text*⟩`}`

The caption is made of `\lstlistingname` followed by a running number, a seperator, and ⟨*caption text*⟩. Either the caption text or, if present, ⟨*short*⟩ will be used for the list of listings.

`label=`⟨*name*⟩

makes a listing referable via `\ref{`⟨*name*⟩`}`.

`\lstlistoflistings`

prints a list of listings. Each entry is with descending priority either the short caption, the caption, the file name or the name of the listing, see also the key `name` in section 4.8.

`nolol=`⟨`true`|`false`⟩ or `nolol`

If true, the listing does not make it into the list of listings.

data `\lstlistlistingname` `Listings`

The header name for the list of listings.

data `\lstlistingname` `Listing`

The caption label for listings.

data `\thelstlisting` `\arabic{lstlisting}`

prints the running number of the caption.

`\lstname`

prints the name of the current listing which is either the file name or the name defined by the `name` key. This command can be used to define a caption or title template, for example by `\lstset{caption=\lstname}`.

`captionpos`=⟨*subset of* `tb`⟩ `t`

specifies the positions of the caption: top and/or bottom of the listing.

`abovecaptionskip`=⟨*dimension*⟩ `\smallskipamount`

`belowcaptionskip`=⟨*dimension*⟩ `\smallskipamount`

is the vertical space above respectively below each caption.

4.10 Margins and line shape

`linewidth`=⟨*dimension*⟩ `\linewidth`

defines the base line width for listings. The following three keys are taken into account additionally.

`xleftmargin`=⟨*dimension*⟩ `0pt`

`xrightmargin`=⟨*dimension*⟩ `0pt`

The dimensions are used as extra margins on the left and right. Line numbers and frames both move respectively shrink or grow accordingly.

`resetmargins`=⟨`true`|`false`⟩ `false`

If true indention from list environments like `enumerate` or `itemize` is reset, i.e. not used.

`breaklines`=⟨`true`|`false`⟩ or `breaklines` `false`

activates or deactivates automatic line breaking of long lines.

`breakatwhitespace`=⟨`true`|`false`⟩ or `breakatwhitespace` `false`

If true, it allows line breaks only at white space.

`prebreak`=⟨*tokens*⟩ `{}`

`postbreak`=⟨*tokens*⟩ `{}`

⟨*tokens*⟩ appear at the end of the current line respectively at the beginning of the next (broken part of the) line.

You must not use dynamic space (in particular spaces) since internally we use `\discretionary`. However `\space` is redefined to be used inside ⟨*tokens*⟩.

breakindent=⟨*dimension*⟩ 20pt

is the indention of the second, third, ... line of broken lines.

breakautoindent=⟨true|false⟩ or breakautoindent true

activates or deactivates automatic indention of broken lines. This indention is used additionally to `breakindent`, see the example below. Visible spaces or visible tabulators might set this auto indention to zero.

In the following example we use tabulators to create long lines, but the verbatim part uses `tabsize=1`.

```
\lstset{postbreak=\space, breakindent=5pt, breaklines}
```

```
"A␣long␣string␣
  is␣broken!"
      "Another␣
        long␣
        line."

{ Now auto
indention is off. }
```

```
\begin{lstlisting}
  "A long string is broken!"
   "Another long line."
\end{lstlisting}

\begin{lstlisting}[breakautoindent
                              =false]
  { Now auto indention is off. }
\end{lstlisting}
```

4.11 Frames

frame=⟨none|leftline|topline|bottomline|lines|single|shadowbox⟩ none

draws either no frame, a single line on the left, at the top, at the bottom, at the top and bottom, a whole single frame, or a shadowbox.

Note that fancyvrb supports the same frame types except `shadowbox`. The shadow color is `rulesepcolor`, see below.

frame=⟨*subset of* trblTRBL⟩ {}

The characters `trblTRBL` are attached to lines at the top and bottom of a listing and to lines on the right and left. Upper case characters are used to draw double rules. So `frame=tlrb` draws a single frame and `frame=TL` double lines at the top and on the left.

Note that frames usually reside outside the listing's space.

frameround=⟨t|f⟩⟨t|f⟩⟨t|f⟩⟨t|f⟩ ffff

The four letters are attached to the top right, bottom right, bottom left and top left corner. In this order. `t` makes the according corner round. If you use round corners, the rule width is controlled via `\thinlines` and `\thicklines`.

Note: The size of the quarter circles depends on `framesep` and is independent of the extra margins of a frame. The size is possibly adjusted to fit LaTeX's circle sizes.

framesep=⟨*dimension*⟩ 3pt

rulesep=⟨*dimension*⟩ 2pt

control the space between frame and listing and between double rules.

framerule=⟨*dimension*⟩ 0.4pt

controls the width of the rules.

framexleftmargin=⟨*dimension*⟩ 0pt

framexrightmargin=⟨*dimension*⟩ 0pt

framextopmargin=⟨*dimension*⟩ 0pt

framexbottommargin=⟨*dimension*⟩ 0pt

are the dimensions which are used additionally to `framesep` to make up the margin of a frame.

backgroundcolor=⟨*color command*⟩

rulecolor=⟨*color command*⟩

fillcolor=⟨*color command*⟩

rulesepcolor=⟨*color command*⟩

specify the colour of the background, the rules, the space between 'text box' and first rule, and of the space between two rules, respectively. Note that the value requires a `\color` command, for example `rulecolor=\color{blue}`.

`frame` does not work with `fancyvrb=true` or when the package internally makes a `\hbox` around the listing! And there are certainly more problems with other commands. Take the time to make a (bug) report.

```
\lstset{framexleftmargin=5mm, frame=shadowbox, rulesepcolor=\color{blue}}
```

```
1  for i:=maxint to 0 do
2  begin
3      { do nothing }
4  end;
```

```
\begin{lstlisting}[numbers=left]
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

Do you want exotic frames? Try the following key if you want for example

```
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

```
\begin{lstlisting}
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

† frameshape={⟨*top shape*⟩}{⟨*left shape*⟩}{⟨*right shape*⟩}{⟨*bottom shape*⟩}

gives you full control over the drawn frame parts. The arguments are not case sensitive.

Both ⟨*left shape*⟩ and ⟨*right shape*⟩ are 'left-to-right' `y`|`n` character sequences (or empty). Each `y` lets the package draw a rule, otherwise the rule is blank. These vertical rules are drawn 'left-to-right' according to the specified shapes. The example above uses `yny`.

⟨*top shape*⟩ and ⟨*bottom shape*⟩ are 'left-rule-right' sequences (or empty). The first 'left-rule-right' sequence is attached to the most inner rule, the second to the next, and so on. Each sequence has three characters: 'rule' is either `y` or `n`; 'left' and 'right' are `y`, `n` or `r` (which makes a corner round). The example uses `RYRYNYYYY` for both shapes: `RYR` describes the most inner (top and bottom) frame shape, `YNY` the middle, and `YYY` the most outer.

To summarize, the example above used

```
\lstset{frameshape={RYRYNYYYY}{yny}{yny}{RYRYNYYYY}}
```

Note that you are not resticted to two or three levels. However you'll get in trouble if you use round corners when they are too big.

4.12 Indexing

`index=[`⟨*number*⟩`][`⟨*keyword classes*⟩`]{`⟨*identifiers*⟩`}`

`moreindex=[`⟨*number*⟩`][`⟨*keyword classes*⟩`]{`⟨*identifiers*⟩`}`

`deleteindex=[`⟨*number*⟩`][`⟨*keyword classes*⟩`]{`⟨*identifiers*⟩`}`

define, add and remove ⟨*identifiers*⟩ and ⟨*keyword classes*⟩ from the index class list ⟨*number*⟩. If you don't specify the optional number, the package assumes ⟨*number*⟩ $= 1$.

Each appearance of the explicitly given identifiers and each appearance of the identifiers of the specified ⟨*keyword classes*⟩ is indexed. For example, you could write `index=[1][keywords]` to index all keywords. Note that `[1]` is required here—otherwise we couldn't use the second optional argument.

`indexstyle=[`⟨*number*⟩`]`⟨*tokens* (*one-parameter command*)⟩ `\lstindexmacro`

⟨*tokens*⟩ actually indexes the identifiers for the list ⟨*number*⟩. In contrast to the style keys, ⟨*tokens*⟩ *must* read exactly one parameter, namely the identifier. Default definition is`\lstindexmacro`

```
\newcommand\lstindexmacro[1]{\index{{\ttfamily#1}}}
```

which you shouldn't modify. Define your own indexing commands and use them as argument to this key.

Section 2.9 describes this feature in detail.

4.13 Column alignment

`columns=[`⟨`c`|`l`|`r`⟩`]`⟨`fixed`|`flexible`|`fullflexible`⟩ `[c]fixed`

selects the respective column format, refer section 2.10.

The optional `c`, `l`, or `r` controls the horizontal orientation of smallest output units (keywords, identifiers, etc.). The arguments work as follows, where vertical bars visualize the effect: | l i s t i n g |, |l i s t i n g |, and | l i s t i n g| in fixed column mode, | listing |, |listing |, and | listing| with flexible columns, and |listing|, |listing|, and |listing| with full flexible columns.

flexiblecolumns=⟨true|false⟩ or flexiblecolumns false

selects the most recently selected flexible or fixed column format, refer to section 2.10.

† keepspaces=⟨true|false⟩ false

keepspaces=true tells the package not to drop spaces to fix column alignment and always converts tabulators to spaces.

basewidth=⟨*dimension*⟩ or

basewidth={⟨*fixed*⟩,⟨*flexible mode*⟩} {0.6em,0.45em}

sets the width of a single character box for fixed and flexible column mode (both to the same value or individually).

fontadjust=⟨true|false⟩ or fontadjust false

If true the package adjusts the base width every font selection. This makes sense only if basewidth is given in font specific units like 'em' or 'ex'—otherwise this boolean has no effect.

After loading the package, it doesn't adjust the width every font selection: it looks at basewidth each listing and uses the value for the whole listing. This is possibly inadequate if the style keys in section 4.6 make heavy font size changes, see the example below.

Note that this key might disturb the column alignment and might have an effect on the keywords' appearance!

```
{ scriptsize font
  doesn't look good }
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

```
\lstset{commentstyle=\scriptsize}
\begin{lstlisting}
{ scriptsize font
  doesn't look good }
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

```
{ scriptsize font
  looks better now }
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

```
\begin{lstlisting}[fontadjust]
{ scriptsize font
  looks better now }
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

4.14 Escaping to LaTeX

Note: *Any escape to LaTeX may disturb the column alignment since the package can't control the spacing there.*

texcl=⟨true|false⟩ or texcl false

activates or deactivates LaTeX comment lines. If activated, comment line delimiters are printed as usual, but the comment line text (up to the end of line) is read as LaTeX code and typeset in comment style.

The example uses C++ comment lines (but doesn't say how to define them). Without `\upshape` we would get *calculate* since the comment style is `\itshape`.

// calculate a_{ij}
A[i][j] = A[j][j]/A[i][j];

```
\begin{lstlisting}[texcl]
// \upshape calculate $a_{ij}$
  A[i][j] = A[j][j]/A[i][j];
\end{lstlisting}
```

mathescape=⟨true|false⟩ false

activates or deactivates special behaviour of the dollar sign. If activated a dollar sign acts as TeX's text math shift.

This key is useful if you want to typeset formulas in listings.

escapechar=⟨*character*⟩ or escapechar={} {}

If not empty the given character escapes the user to LaTeX: all code between two such characters is interpreted as LaTeX code. Note that TeX's special characters must be entered with a preceding backslash, e.g. `escapechar=\%`.

escapeinside=⟨*character*⟩⟨*character*⟩ or escapeinside={} {}

Is a generalization of `escapechar`. If the value is not empty, the package escapes to LaTeX between the first and second character.

escapebegin=⟨*tokens*⟩ {}

escapeend=⟨*tokens*⟩ {}

The tokens are executed at the beginning respectively at the end of each escape, in particular for `texcl`. See section 7 for an application.

// *calculate* a_{ij}
$a_{ij} = a_{jj}/a_{ij}$;

```
\begin{lstlisting}[mathescape]
// calculate $a_{ij}$
  $a_{ij} = a_{jj}/a_{ij}$;
\end{lstlisting}
```

// *calc*ulate a_{ij}
$a_{ij} = a_{jj}/a_{ij}$;

```
\begin{lstlisting}[escapechar=\%]
// calc%ulate $a_{ij}$%
  %$a_{ij} = a_{jj}/a_{ij}$%;
\end{lstlisting}
```

// *calc*ulate a_{ij}
$a_{ij} = a_{jj}/a_{ij}$;

```
\lstset{escapeinside=`'}
\begin{lstlisting}
// calc`ulate $a_{ij}$'
  `$a_{ij} = a_{jj}/a_{ij}$';
\end{lstlisting}
```

In the first example the comment line up to a_{ij} has been typeset by the listings package in comment style. The a_{ij} itself is typeset in 'TEX math mode' without comment style. About the half comment line of the second example has been typeset by this package. The rest is in 'LATEX mode'.

To avoid problems with the current and future version of this package:

1. Don't use any command of the listings package when you have escaped to LATEX.
2. Any environment must start and end inside the same escape.
3. You might use `\def`, `\edef`, etc., but do not assume that the definitions are present later—except they are `\global`.
4. `\if \else \fi`, groups, math shifts `$` and `$$`, ... must be balanced each escape.
5. ...

Expand that list yourself and mail me about new items.

4.15 Interface to fancyvrb

The fancyvrb package—fancy verbatims—from Timothy van Zandt provides macros for reading, writing and typesetting verbatim code. It has some remarkable features the listings package doesn't have. (Some are possible, but you must find somebody who implements them ; –).

`fancyvrb`=⟨`true`|`false`⟩

> activates or deactivates the interface. If active, verbatim code is read by fancyvrb but typeset by listings, i.e. with emphasized keywords, strings, comments, and so on. Internally we use a very special definition of `\FancyVerbFormatLine`.
>
> This interface works with `Verbatim`, `BVerbatim` and `LVerbatim`. But you shouldn't use fancyvrb's `defineactive`. (As far as I can see it doesn't matter since it does nothing at all, but for safety ...) If fancyvrb and listings provide similar functionality, you should use fancyvrb's.

`fvcmdparams`=⟨*command*$_1$⟩⟨*number*$_1$⟩... `\overlay1`

`morefvcmdparams`=⟨*command*$_1$⟩⟨*number*$_1$⟩...

> If you use fancyvrb's `commandchars`, you must tell the listings package how many arguments each command takes. If a command takes no arguments, there is nothing to do.
>
> The first (third, fifth, ...) parameter to the keys is the command and the second (fourth, sixth, ...) is the number of arguments that command takes. So, if you want to use `\textcolor{red}{keyword}` with the fancyvrb-listings interface, you should write `\lstset{morefvcmdparams=\textcolor 2}`.

First verbatim line.
[Second] verbatim line.

```
\lstset{morecomment=[l]\ }% :-)
\fvset{commandchars=\\\{\}}

\begin{BVerbatim}
First verbatim line.
\fbox{Second} verbatim line.
\end{BVerbatim}

\par\vspace{72.27pt}
```

First *verbatim line.*
[Second] *verbatim line.*

```
\lstset{fancyvrb}
\begin{BVerbatim}
First verbatim line.
\fbox{Second} verbatim line.
\end{BVerbatim}
\lstset{fancyvrb=false}
```

The lines typeset by the listings package are wider since the default `basewidth` equals not the width of a single typewriter type character. Moreover note that the first space begins a comment as defined at the beginning of the example.

4.16 Environments

If you want to define your own pretty-printing environments, try the following command. The syntax comes from LaTeX's `\newenvironment`.

`\lstnewenvironment`
  `{`⟨*name*⟩`}[`⟨*number*⟩`][`⟨*opt. default arg.*⟩`]`
  `{`⟨*starting code*⟩`}`
  `{`⟨*ending code*⟩`}`

As a simple example we could just select a particular language.

```
\lstnewenvironment{pascal}
    {\lstset{language=pascal}}
    {}
```

for i:=**maxint** **to** 0 **do**
begin
  { *do nothing* }
end;

```
\begin{pascal}
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{pascal}
```

Doing other things is as easy, for example, using more keys and adding an optional argument to adjust settings each listing:

```
\lstnewenvironment{pascalx}[1][]
    {\lstset{language=pascal,numbers=left,numberstyle=\tiny,float,#1}}
    {}
```

4.17 Language definitions

You should first read section 3.2 for an introduction to language definitions. Otherwise you're probably unprepared for the full syntax of `\lstdefinelanguage`.

`\lstdefinelanguage`
`[[`⟨*dialect*⟩`]]{`⟨*language*⟩`}`
`[[`⟨*base dialect*⟩`]{`⟨*and base language*⟩`}]`
`{`⟨*key=value list*⟩`}`
`[[`⟨*list of required aspects (keywordcomments,texcs,etc.)*⟩`]]`

defines the (given dialect of the) programming language ⟨*language*⟩. If the language definition is based on another definition, you must specify the whole `[`⟨*base dialect*⟩`]{`⟨*and base language*⟩`}`. Note that an empty ⟨*base dialect*⟩ uses the default dialect!

The last optional argument should specify all required aspects. This is a delicate point since the aspects are described in the developer's guide. You might use existing languages as templates. For example, ANSI C uses keywords, comments, strings and directives.

`\lst@definelanguage` has the same syntax and is used to define languages in the driver files.

→ Where should I put my language definition? If you need the language for one particular document, put it into the preamble of that document. Otherwise create the local file '`lstlang0.sty`' or add the definition to that file, but use '`\lst@definelanguage`' instead of '`\lstdefinelanguage`'. However, you might want to send the definition to the address in section 2.1. Then it will be published under the LaTeX Project Public License.

`\lstalias{`⟨*alias*⟩`}{`⟨*language*⟩`}`

defines an alias for a programming language. Each ⟨*alias*⟩ is redirected to the same dialect of ⟨*language*⟩. It's also possible to define an alias for one particular dialect only:

`\lstalias[`⟨*alias dialect*⟩`]{`⟨*alias*⟩`}[`⟨*dialect*⟩`]{`⟨*language*⟩`}`

Here all four parameters are *nonoptional* and an alias with empty ⟨*dialect*⟩ will select the default dialect. Note that aliases can't be nested: The two aliases '`\lstalias{foo1}{foo2}`' and '`\lstalias{foo2}{foo3}`' redirect `foo1` *not* to `foo3`.

All remaining keys in this section are intended to build language definitions. *No other key should be used in such a definition!*

Keywords We begin with keyword building keys. Note: *If you want to enter* `\`, `{`, `}`, `%`, `#` *or* `&` *inside or as an argument here or below, you must do it with a preceding backslash!*

†bug `keywordsprefix=`⟨*prefix*⟩

All identifiers starting with ⟨*prefix*⟩ will be printed as first order keywords.

Bugs: Currently there are several limitations. (1) The prefix is always case sensitive. (2) Only one prefix can be defined at the same time. (3) If used 'standalone', the key might work only after selecting a nonempty language (and switching back to the empty language if necessary). (4) The key does not respect the value of `classoffset` and has no optional class ⟨*number*⟩ argument.

`keywords=[`⟨*number*⟩`]{`⟨*list of keywords*⟩`}`

`morekeywords=[`⟨*number*⟩`]{`⟨*list of keywords*⟩`}`

`deletekeywords=[`⟨*number*⟩`]{`⟨*list of keywords*⟩`}`

define, add to or remove the keywords from keyword list ⟨*number*⟩. The use of `keywords` is discouraged since it deletes all previously defined keywords in the list and is thus incompatible with the `alsolanguage` key.

Please note the keys `alsoletter` and `alsodigit` below if you use unusual charaters in keywords.

deprecated `ndkeywords={`⟨*list of keywords*⟩`}`

deprecated `morendkeywords={`⟨*list of keywords*⟩`}`

deprecated `deletendkeywords={`⟨*list of keywords*⟩`}`

define, add to or remove the keywords from keyword list 2. The use of `ndkeywords` is strongly discouraged.

addon,optional `texcs=[`⟨*class number*⟩`]{`⟨*list of control sequences (without backslashes)*⟩`}`

addon,optional `moretexcs=[`⟨*class number*⟩`]{`⟨*list of control sequences (without backslashes)*⟩`}`

addon,optional `deletetexcs=[`⟨*class number*⟩`]{`⟨*list of control sequences (without backslashes)*⟩`}`

Ditto for control sequences in TeX and LaTeX.

optional `directives={`⟨*list of compiler directives*⟩`}`

optional `moredirectives={`⟨*list of compiler directives*⟩`}`

optional `deletedirectives={`⟨*list of compiler directives*⟩`}`

defines compiler directives in C, C++, Objective-C, and POV.

`sensitive=`⟨`true`|`false`⟩

makes the keywords, control sequences, and directives case sensitive and insensitive, respectively. This key affects the keywords, control sequences, and directives only when a listing is processed. In all other situations they are case sensitive, for example, `deletekeywords={save,Test}` removes 'save' and 'Test', but neither 'SavE' nor 'test'.

`alsoletter={`⟨*character sequence*⟩`}`

`alsodigit={`⟨*character sequence*⟩`}`

`alsoother={`⟨*character sequence*⟩`}`

All identifiers (keywords, directives, and such) begin with a letter and goes on with alpha-numeric characters (letters and digits). For example, if you write `keywords={one-two,\#include}`, the minus must become a digit and the sharp a letter since the keywords can't be detected otherwise.

Table 2 show the standard configuration of the listings package. The three keys overwrite the default behaviour. Each character of the sequence becomes a letter, digit and other, respectively.

Table 2: Standard character table

<table>
<tr><td>class</td><td>characters</td></tr>
<tr><td>letter</td><td><code>A B C D E F G H I J K L M N O P Q R S T U V W X Y Z</code>
<code>a b c d e f g h i j k l m n o p q r s t u v w x y z</code>
<code>@ $ _</code></td></tr>
<tr><td>digit</td><td><code>0 1 2 3 4 5 6 7 8 9</code></td></tr>
<tr><td>other</td><td><code>! " # % & ' ( ) * + , - . / : ; < = > ?</code>
<code>[\ ] ^ { | } ~</code></td></tr>
<tr><td>space</td><td>chr(32)</td></tr>
<tr><td>tabulator</td><td>chr(9)</td></tr>
<tr><td>form feed</td><td>chr(12)</td></tr>
</table>

Note: Extended characters of codes 128–255 (if defined) are *currently* letters.

`otherkeywords={`⟨*keywords*⟩`}`

Each given 'keyword' is printed in keyword style, but without changing the 'letter', 'digit' and 'other' status of the characters. This key is designed to define keywords like `=>`, `->`, `-->`, `--`, `::`, and so on. If one keyword is a subsequence of another (like `--` and `-->`), you must specify the shorter first.

renamed,optional `tag=`⟨*character*⟩⟨*character*⟩ or `tag={}`

The first order keywords are active only between the first and second character. This key is used for HTML.

Strings

`string=[`⟨`b`|`d`|`m`|`bd`⟩`]{`⟨*delimiter* (*character*)⟩`}`

`morestring=[`⟨`b`|`d`|`m`|`bd`⟩`]{`⟨*delimiter*⟩`}`

`deletestring=[`⟨`b`|`d`|`m`|`bd`⟩`]{`⟨*delimiter*⟩`}`

define, add to or delete the delimiter from the list of string delimiters. Starting and ending delimiters are the same, i.e. in the source code the delimiters must match each other.

The optional argument is the type and controls how the delimiter itself is represented in a string or character literal: it is escaped by a `b`ackslash, `d`oubled (or both is allowed via `bd`) or it is '`m`atlabed'. The latter one is a special type for Ada and Matlab and possibly more languages where the string delimters are also used for other purposes. In general the delimiter is also doubled, but a string does not start after a letter, a right parenthesis, or a right bracket.

Comments

`comment=[`⟨*type*⟩`]`⟨*delimiter(s)*⟩

`morecomment=[`⟨*type*⟩`]`⟨*delimiter(s)*⟩

`deletecomment=[`⟨*type*⟩`]`⟨*delimiter(s)*⟩

Ditto for comments, but some types require more than a single delimiter. The following overview uses `morecomment` as the only example.

`morecomment=[l]`⟨*delimiter*⟩

The delimiter starts a comment line, which in general starts with the delimiter and ends at end of line. If the character sequence `//` should start a comment line (like in C++, Comal 80 or Java), `morecomment=[l]//` is the correct declaration. For Matlab it would be `morecomment=[l]\%`—note the preceding backslash.

`morecomment=[s]{`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}`

Here we have two delimiters. The second ends a comment starting with the first delimiter. If you require two such comments you can use this type twice. C, Java, PL/I, Prolog and SQL all define single comments via `morecomment=[s]{/*}{*/}`, and Algol does it with `morecomment=[s]{\#}{\#}`, which means that the sharp delimits both beginning and end of a single comment.

`morecomment=[n]{`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}`

is similar to type `s`, but comments can be nested. Identical arguments are not allowed—think a while about it! Modula-2 and Oberon-2 use `morecomment=[n]{(*}{*)}`.

`morecomment=[f][][`⟨*n=preceding columns*⟩`]`⟨*delimiter*⟩

The delimiter starts a comment line if and only if it appears on a fixed column-number, namely if it is in column n (zero based).

optional `keywordcomment={`⟨*keywords*⟩`}`

optional `morekeywordcomment={`⟨*keywords*⟩`}`

optional `deletekeywordcomment={`⟨*keywords*⟩`}`

A keyword comment begins with a keyword and ends with the same keyword. Consider `keywordcomment={comment,co}`. Then '**comment**...**comment**' and '**co**...**co**' are comments.

optional `keywordcommentsemicolon={`⟨*keywords*⟩`}{`⟨*keywords*⟩`}{`⟨*keywords*⟩`}`

The definition of a 'keyword comment semicolon' requires three keyword lists, e.g. `{end}{else,end}{comment}`. A semicolon always ends such a comment. Any keyword of the first argument begins a comment and any keyword of the second argument ends it (and a semicolon also); a comment starting with any keyword of the third argument is terminated with the next semicolon only. In the example all possible comments are '**end**...**else**', '**end**...**end**' (does not start a comment again) and '**comment**...;' and '**end**...;'. Maybe a curious definition, but Algol and Simula use such comments.

Note: The keywords here need not to be a subset of the defined keywords. They won't appear in keyword style if they aren't.

optional `podcomment=`⟨`true`|`false`⟩

activates or deactivates PODs—Perl specific.

4.18 Installation

Software installation

1. Following the TeX directory structure (TDS), you should put the files of the listings package into directories as follows:

`listings.pdf`	→	`texmf/doc/latex/listings`
`listings.dtx`, `listings.ins`, `listings.ind`, `lstpatch.sty`, `lstdrvrs.dtx`	→	`texmf/source/latex/listings`

   Note that you possibly don't have a patch file `lstpatch.sty`. If you don't use the TDS, simply adjust the directories below.

2. Create the directory `texmf/tex/latex/listings` or remove all files except `lst`⟨*whatever*⟩`0.sty` and `lstlocal.cfg` from that directory.

3. Change the working directory to `texmf/source/latex/listings` and run `listings.ins` through TeX.

4. Move the generated files to `texmf/tex/latex/listings` if this is not already done.

`listings.sty`, `lstmisc.sty`,		(kernel and add-ons)
`listings.cfg`,		(configuration file)
`lstlang`⟨*number*⟩`.sty`,		(language drivers)
`lstpatch.sty`	→	`texmf/tex/latex/listings`

5. If your TeX implementation uses a file name database, update it.

6. If you receive a patch file later on, put it where `listings.sty` is (and update file name database).

Note that listings requires at least version 1.10 of the keyval package included in the graphics bundle by David Carlisle.

Software configuration Read this only if you encounter problems with the standard configuration or if you want the package to suit foreign languages, for example.

Never modify a file from the listings package, in particular not the configuration file. Each new installation or new version overwrites it. The software license allows modification, but I can't recommend it. It's better to create one or more of the files

`lstmisc0.sty`	for	local add-ons (see developer's guide),
`lstlang0.sty`	for	local language definitions (see 4.17), and
`lstlocal.cfg`	as	local configuration file

and put it/them to the other listings files. These three files are not touched by a new installation except you remove them. If `lstlocal.cfg` exists, it is loaded after `listings.cfg`. You might want to change one of the following parameters.

data `\lstaspectfiles` contains `lstmisc0.sty,lstmisc.sty` *data* `\lstlanguagefiles` contains `lstlang0.sty,lstlang1.sty,lstlang2.sty,lstlang3.sty`

The package uses the specified files to find add-ons and language definitions.

Moreover you might want to adjust `\lstlistlistingname`, `\lstlistingname`, `defaultdialect`, `\lstalias`, or `\lstalias`as described in earlier section.

5 Experimental features

This section describes the more or less unestablished parts of this package. It's unlikely that they are all removed (except it is stated explicitly), but they are liable to (heavy) changes and improvements. Such features have been †-marked in the last sections. So, if you find anything †-marked here, you should be very, very careful.

5.1 Listings inside arguments

There are some things to consider if you want to use `\lstinline` or the listing environment inside arguments. Since TeX reads the argument before the '`lst`-macro' is executed, this package can't do anything to preserve the input: spaces shrink to one space, the tabulator and the end of line are converted to spaces, TeX's comment character is not printable, and so on. Hence, *you* must work a bit more. You have to put a backslash in front of each of the following four characters: `\{}%`. Moreover you must protect spaces in the same manner if: (i) there are two or more spaces following each other or (ii) the space is the first character in the line. That's not enough: Each line must be terminated with a 'line feed' `^^J`. And you can't escape to LaTeX inside such listings!

The easiest examples are with `\lstinline` since we need no line feed.

```
\footnote{\lstinline{var i:integer;} and
          \lstinline!protected\ \ spaces! and
          \fbox{\lstinline!\\\{\}\%!}}
```

yields[1] if the current language is Pascal. Note that this example shows another experimental feature: use of argument braces as delimiters. This is described in section 4.2.

And now an environment example:

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
```

```
\fbox{%
\begin{lstlisting}^^J
\ !"#$\%&'()*+,-./^^J
0123456789:;<=>?^^J
@ABCDEFGHIJKLMNO^^J
PQRSTUVWXYZ[\\]^_^^J
`abcdefghijklmno^^J
pqrstuvwxyz\{|\}~^^J
\end{lstlisting}}
```

→ You might wonder that this feature is still experimental. The reason: You shouldn't use listings inside arguments; it's not always safe.

[1] **var** i:**integer**; and protected spaces and $\boxed{\texttt{\textbackslash\{\}\%}}$

5.2 † Export of identifiers

It would be nice to export function or procedure names. In general that's a dream so far. The problem is that programming languages use various syntaxes for function and procedure declaration or definition. A general interface is completely out of the scope of this package—that's the work of a compiler and not of a pretty-printing tool. However, it is possible for particular languages: in Pascal each function or procedure definition and variable declaration is preceded by a particular keyword. Note that you must request the following keys with `procnames` option: `\usepackage[procnames]{listings}`.

†optional `procnamekeys={`⟨*keywords*⟩`}` `{}`

†optional `moreprocnamekeys={`⟨*keywords*⟩`}`

†optional `deleteprocnamekeys={`⟨*keywords*⟩`}`

each specified keyword indicates a function or procedure definition. Any identifier following such a keyword appears in 'procname' style. For Pascal you might use

```
procnamekeys={program,procedure,function}
```

†optional `procnamestyle=`⟨*style*⟩ `keywordstyle`

defines the style in which procedure and function names appear.

†optional `indexprocnames=`⟨`true`|`false`⟩ `false`

If activated, procedure and function names are also indexed.

To do: The procnames aspect is unsatisfactory (since unchanged for more than four years). It marks and indexes the function definitions so far, but it would be possible to mark also the following function calls, for example. A key could control whether function names are added to a special keyword class, which then appears in 'procname' style. But should these names be added globally? There are good reasons for both. Of course, we would also need a key to reset the name list.

5.3 † Hyper references

This very small aspect must be requested via `hyper` option since it is experimental. One perspective for the future is to combine this aspect with procnames. Then it should be possible to click on a function name and jump to its definition, for example.

†optional `hyperref={`⟨*identifiers*⟩`}`

†optional `morehyperref={`⟨*identifiers*⟩`}`

†optional `deletehyperref={`⟨*identifiers*⟩`}`

Hyper references the specified identifiers (via hyperref package). A 'click' on such an identifier jumps to the previous occurrence.

†optional `hyperanchor=`⟨*two-parameter macro*⟩ `\hyper@@anchor`

†optional `hyperlink=`⟨*two-parameter macro*⟩ `\hyperlink`

The macros are used to set an hyper anchor and link, respectively. The defaults are suited for the hyperref package.

5.4 Literate programming

We begin with an example and hide the crucial key=value list.

var i:**integer**;	`\begin{lstlisting}`
	`var i:integer;`
if (i$\leq$0) i $\leftarrow$ 1;	`if (i<=0) i := 1;`
if (i$\geq$0) i $\leftarrow$ 0;	`if (i>=0) i := 0;`
if (i$\neq$0) i $\leftarrow$ 0;	`if (i<>0) i := 0;`
	`\end{lstlisting}`

Funny, isn't it? We could write `i := 0` respectively i $\leftarrow$ 0 instead, but that's not literate!Now you might want to know how this has been done. Have a *close* look at the following key.

† `literate=`⟨*replacement item*⟩...⟨*replacement item*⟩

First note that there are no commas between the items. Each item consists of three arguments: `{`⟨*replace*⟩`}{`⟨*replacement text*⟩`}{`⟨*length*⟩`}`. ⟨*replace*⟩ is the original character sequence. Instead of printing these characters, we use ⟨*replacement text*⟩, which takes the width of ⟨*length*⟩ characters in the output.

Each 'printing unit' in ⟨*replacement text*⟩ *must* be braced except it's a single character. For example, you must put braces around `$\leq$`. If you want to replace `<-1->` by `$\leftarrow1\rightarrow$`, the replacement item would be `{<-1->}{{$\leftarrow$}1{$\rightarrow$}}3`. Note the braces around the arrows.

If one ⟨*replace*⟩ is a subsequence of another ⟨*replace*⟩, you must use the shorter sequence first. For example, `{-}` must be used before `{--}` and this before `{-->}`.

In the example above I've used

```
literate={:=}{{$\gets$}}1 {<=}{{$\leq$}}1 {>=}{{$\geq$}}1 {<>}{{$\neq$}}1
```

To do: Of course, it's good to have keys for adding and removing single ⟨*replacement item*⟩s. Maybe the key(s) should work in the same fashion as the string and comment definitions, i.e. one item per key=value. This way it would be easier to provide better auto-detection in case of a subsequence.

5.5 LGrind definitions

Yes, it's a nasty idea to steal language definitions from other programs. Nevertheless, it's possible for the LGrind definition file—at least partially. Please note that this file must be found by TEX.

optional `lgrindef=`⟨*language*⟩

scans the `lgrindef` language definition file for ⟨*language*⟩ and activates it if present. Note that not all LGrind capabilities have a listings analogue.

Note that 'Linda' language doesn't work properly since it defines compiler directives with preceding '`#`' as keywords.

data,optional `\lstlgrindeffile` `lgrindef.`

contains the (path and) name of the definition file.

5.6 † Automatic formatting

The automatic source code formatting is far away from being good. First of all, there are no general rules on how source code should be formatted. So 'format definitions' must be flexible. This flexibility requires a complex interface, a powerful 'format definition' parser, and lots of code lines behind the scenes. Currently, format definitions aren't flexible enough (possibly not the definitions but the results). A single 'format item' has the form

⟨*input chars*⟩`=[`⟨*exceptional chars*⟩`]`⟨*pre*⟩`[`⟨`\string`⟩`]`⟨*post*⟩

Whenever ⟨*input chars*⟩ aren't followed by one of the ⟨*exceptional chars*⟩, formatting is done according to the rest of the value. If `\string` isn't specified, the input characters aren't printed (except it's an identifier or keyword). Otherwise ⟨*pre*⟩ is 'executed' before printing the original character string and ⟨*post*⟩ afterwards. These two are 'subsets' of

- `\newline` —ensuring a new line;
- `\space` —ensuring a whitespace;
- `\indent` —increasing indention;
- `\noindent` —descreasing indention.

Now we can give an example.

```
\lstdefineformat{C}{%
    \{=\newline\string\newline\indent,%
    \}=\newline\noindent\string\newline,%
    ;=[\ ]\string\space}
```

```
for (int i=0; i<10; i++)
{
    /* wait */
}
;
```

```
\begin{lstlisting}[format=C]
for (int i=0;i<10; i++){/* wait */};
\end{lstlisting}
```

Not good. But there is a (too?) simple work-around:

```
\lstdefineformat{C}{%
    \{=\newline\string\newline\indent,%
    \}=[;]\newline\noindent\string\newline,%
    \};=\newline\noindent\string\newline,%
    ;=[\ ]\string\space}
```

```
for (int i=0; i<10; i++)
{
    /* wait */
};
```

```
\begin{lstlisting}[format=C]
for (int i=0;i<10; i++){/* wait */};
\end{lstlisting}
```

Sometimes the problem is just to find a suitable format definition. Further formatting is complicated. Here are only three examples with increasing level of difficulty.

1. Insert horizontal space to separate function/procedure name and following parenthesis or to separate arguments of a function, e.g. add the space after a comma (if inside function call).
2. Smart breaking of long lines. Consider long 'and/or' expressions. Formatting should follow the logical structure!
3. Context sensitive formatting rules. It can be annoying if empty or small blocks take three or more lines in the output—think of scrolling down all the time. So it would be nice if the block formatting was context sensitive.

Note that this is a very first and clumsy attempt to provide automatic formatting—clumsy since the problem isn't trivial. Any ideas are welcome. Implementations also. Eventually you should know that you must request format definitions at package loading, e.g. via `\usepackage[formats]{listings}`.

5.7 Arbitrary linerange markers

Instead of using `linerange` with line numbers, one can use text markers. Each such marker consists of a ⟨*prefix*⟩, a ⟨*text*⟩, and a ⟨*suffix*⟩. You once (or more) define prefixes and suffixes and then use the marker text instead of the line numbers.

```
\lstset{rangeprefix=\{\ ,% curly left brace plus space
        rangesuffix=\ \}}% space plus curly right brace
```

```
{ loop 2 }
for i:=maxint to 0 do
begin
    { do nothing }
end;
{ end }
```

```
\begin{lstlisting}%
      [linerange=loop\ 2-end]
{ loop 1 }
for i:=maxint to 0 do
begin
    { do nothing }
end;
{ end }
{ loop 2 }
for i:=maxint to 0 do
begin
    { do nothing }
end;
{ end }
\end{lstlisting}
```

Note that TEX's special characters like the curly braces, the space, the percent sign, and such must be escaped with a backslash.

`rangebeginprefix=`⟨*prefix*⟩

`rangebeginsuffix=`⟨*suffix*⟩

`rangeendprefix=`⟨*prefix*⟩

`rangeendsuffix=`⟨*suffix*⟩

define individual prefixes and suffixes for the begin- and end-marker.

`rangeprefix=`⟨*prefix*⟩

rangesuffix=⟨*suffix*⟩

define identical prefixes and suffixes for the begin- and end-marker.

includerangemarker=⟨true|false⟩ true

shows or hides the markers in the output.

```
for i:=maxint to 0 do
begin
    { do nothing }
end;
```

```
\begin{lstlisting}%
      [linerange=loop\ 1-end,
       includerangemarker=false,
       frame=single]
{ loop 1 }
for i:=maxint to 0 do
begin
    { do nothing }
end;
{ end }
\end{lstlisting}
```

Tips and tricks

Note: This part of the documentation is under construction. Section 7 must be sorted by topic and ordered in some way. Moreover a new section 'Examples' is planned, but not written. Lack of time is the main problem ...

6 Troubleshooting

If you're faced with a listings' package problem, there are some steps you should undergo before you make a bug report. First you should consult the reference guide whether the problem is already known. If not, create a *minimal* file which reproduces the problem. Follow these instructions:

1. Start from the minimal file in section 1.1.
2. Add the LaTeX code which causes the problem, but keep it short. In particular, keep the number of additional packages small.
3. Remove some code from the file (and the according packages) until the problem disappears. Then you've found a crucial piece.
4. Add this piece of code again and start over with step 3 until all code and all packages are substantial.
5. You now have a minimal file. Send a bug report to the address on the first page of this documentation and include the minimal file together with the created `.log`-file. If you use a very special package (i.e. not on CTAN), also include the package if its software license allows it.

7 How tos

How to reference line numbers

You want to put `\label{`⟨*whatever*⟩`}` into a LaTeX escape which is inside a comment whose delimiters aren't printed? The compiler won't see the LaTeX code since inside a comment, and the listings package won't print anything since the delimiters are dropped and `\label` doesn't produce any printable output. Well, your wish is granted.

In Pascal, for example, you could make the package recognize the 'special' comment delimiters `(*@` and `@*)` as begin-escape and end-escape sequences. Then you can use this special comment for `\label`s and other things.

```
for i:=maxint to 0 do
begin
    { comment }
end;
```

Line 3 shows a comment.

```
\lstset{escapeinside={(*@}{@*)}}

\begin{lstlisting}
for i:=maxint to 0 do
begin
    { comment }(*@\label{comment}@*)
end;
\end{lstlisting}
Line \ref{comment} shows a comment.
```

→ Can I use '`(*@`' and '`*)`' instead? Yes.

→ Can I use '`(*`' and '`*)`' instead? Sure. If you want this.

→ Can I use '`{@`' and '`@}`' instead? No, never! The second delimiter is not allowed. The character '`@`' is defined to check whether the escape is over. But reading the lonely 'end-argument' brace, TeX encounters the error '`Argument of @ has an extra }`'. Sorry.

→ Can I use '`{`' and '`}`' instead? No. Again the second delimiter is not allowed. Here now TeX would give you a '`Runaway argument`' error. Since '`}`' is defined to check whether the escape is over, it won't work as 'end-argument' brace.

→ And how can I use a comment line? For example, write '`escapeinside={//*}{\^^M}`'. Here `\^^M` represents the end of line character.

How to gobble characters

To make your LaTeX code more readable, you might want to indent your `lstlisting` listings. This indention must be removed for pretty-printing. If you indent each code line by three characters, you can remove them via `gobble=3`:

```
for i:=maxint to 0 do
begin
    { do nothing }
end;

Write('Case␣insensitive␣');
WritE('Pascal␣keywords.');
```

```
\begin{lstlisting}[gobble=3]
1␣␣for␣i:=maxint␣to␣0␣do
␣2␣begin
␣␣3␣␣␣␣{␣do␣nothing␣}
123end;

␣␣␣Write('Case␣insensitive␣');
␣␣␣WritE('Pascal␣keywords.');
\end{lstlisting}
```

Note that empty lines as well as the beginning and the end of the environment need not to respect the indention. But never indent the end by more than '`gobble`' characters. Moreover note that tabulators expand to `tabsize` spaces before we gobble.

→ Could I use 'gobble' together with '\lstinputlisting'? Yes, but it has no effect.

→ Note that 'gobble' can also be set via '\lstset'.

How to include graphics

Herbert Weinhandl found a very easy way to include graphics in listings. Thanks for contributing this idea—an idea I never have had.

Some programming languages allow the dollar sign to be part of an identifier. But except for intermediate function names or library functions, this character is most often unused. The listings package defines the `mathescape` key, which lets '`$`' escape to TeX's math mode. This makes the dollar character an excellent candidate for our purpose here: use a package which can include a graphic, set `mathescape` true, and include the graphic between two dollar signs, which are inside a comment.

The following example is originally from a header file I got from Herbert. For the presentation here I use the `lstlisting` environment and an excerpt from the header file. The `\includegraphics` command is from David Carlisle's graphics bundle.

```
\begin{lstlisting}[mathescape=true]
/*
 $ \includegraphics[height=1cm]{defs-p1.eps} $
 */
typedef struct {
  Atom_T          *V_ptr;   /* pointer to Vacancy in grid    */
  Atom_T          *x_ptr;   /* pointer to (A|B) Atom in grid */
} ABV_Pair_T;
\end{lstlisting}
```

The result looks pretty good. Unfortunately you can't see it.

How to get closed frames on each page

The package supports closed frames only for listings which don't cross pages. If a listing is split on two pages, there is neither a bottom rule at the bottom of a page, nor a top rule on the following page. If you insist on these rules, you might want to use `framed.sty` by Donald Arseneau. Then you could write

```
\begin{framed}
\begin{lstlisting}
  or \lstinputlisting{...}
\end{lstlisting}
\end{framed}
```

The package also provides a `shaded` environment. If you use it, you shouldn't forget to define `shadecolor` with the color package.

How to print national characters with Λ and listings

Apart from typing in national characters directly, you can use the 'escape' feature described in section 4.14. The keys `escapechar`, `escapeinside`, and `texcl` allow partial usage of LaTeX code.

Now, if you use Λ (Lambda, the LaTeX pendant to Omega) and want, for example, Arabic comment lines, you need not to write `\begin{arab}` ... `\end{arab}` each escaped comment line. This can be automated:

```
\lstset{escapebegin=\begin{arab},escapeend=\end{arab}}

\begin{lstlisting}[texcl]
// Replace text by Arabic comment.
for (int i=0; i<1; i++) { };
\end{lstlisting}
```

If your programming language doesn't have comment lines, you'll have to use `escapechar` or `escapeinside`:

```
\lstset{escapebegin=\begin{greek},escapeend=\end{greek}}

\begin{lstlisting}[escapeinside=`']
/* `Replace text by Greek comment.' */
for (int i=0; i<1; i++) { };
\end{lstlisting}
```

Note that the delimiters ` and ' are essential here. The example doesn't work without them. There is a more clever way if the comment delimiters of the programming language are single characters like the braces in Pascal:

```
\lstset{escapebegin=\textbraceleft\begin{arab},
        escapeend=\end{arab}\textbraceright}

\begin{lstlisting}[escapeinside=\{\}]
for i:=maxint to 0 do
begin
    { Replace text by Arabic comment. }
end;
\end{lstlisting}
```

Please note that the 'interface' to Λ is completely untested. Reports are welcome!

How to get bold typewriter type keywords

Use the LuxiMono package.

How to get the developer's guide

In the *source directory* of the listings package, i.e. where the `.dtx` files are, create the file `ltxdoc.cfg` with the following contents.

```
\AtBeginDocument{\AlsoImplementation}
```

Then run `listings-1.3.dtx` through LaTeX twice, run Makeindex, and one last time LaTeX on `listings-1.3.dtx`. This creates the whole documentation including User's guide, Reference guide, Developer's guide, and Implementation.

Index