The Listings Package

Carsten Heinz <cheinz@gmx.de>

2000/08/23 Version 0.21

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

Trademarks Trademarks appear throughout this documentation without any trademark symbol. So you can't assume that names are free. There is no intention of infringement; the usage is to the benefit of the trademark owner.

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

Please note: In this release I haven't cared much about the reference guide. So some information might be old.

News and changes The features 'breaklines' and 'index' aren't experimental any longer. Both's functionality have been extended (but I removed some indexing keys which are now obsolete). 'emph' is new and a collection of keyword classes. Its introduction should solve all highlighting problems: you can specify hundreds of different identifier lists and styles—if TEX's memory suffices.

I've added the experimental features 'hyper' (references) and 'lgrind' (language definitions), and some other keys labelled as *new* as usual. The user's guide has been totally rewritten, but neither the developer's guide nor the documentation of implementation is up-to-date.

The commands and keys `\lststorekeywords`, `pre`, `post`, `\lstname` and `\lstintname` have been removed. They are obsolete since version 0.20. But I plan to reintroduce some funtionality via `everydisplay` and `everytext`, which are undefined yet.

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 *2.13* .

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}% activate Pascal
% Examples can be inserted 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? Read section 2.2 on the installation process. If this doesn't help, you should consult your system administrator and/or the local TeX and LaTeX guides.

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

1.2 Typesetting listings

Three types of source codes are supported: code snippets inside paragraphs and code segments or listings of stand alone files as separate paragraphs. The difference between inside and separate paragraph is the same as between text style and display style formulas.

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, i.e. `\lstinline$var i:integer;$` gives the same result.

→ Don't even try the code as an argument: '`\lstinline{var i:integer;}`' *will not work*.

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 be 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`'.

→ Okay, but it's still not true. The environment takes an argument and all these arguments on the next few pages are empty. It's a name argument and has something to do with line numbers.

→ I see—I've read the section about line numbers. Wouldn't it be easier to define another environment for that purpose? No, no, no. Never do that again! Read this *Getting started* section from the beginning to the end. However, here is an answer to your question: No, it's a matter of taste.

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}[first=2,last=5]{}
for i:=maxint to 0 do
begin
    { do nothing }
end;

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

→ Hold on! I've several questions. Where comes the frame from and what is it good for? You can put frames around all listings except code fragments. 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 4.3. Warning: First read ahead on how to use keys in general.

→ I get obscure error messages when using '`first`'. That shouldn't happen. Make a bug report as described in section *3.1 Troubleshooting*.

Stand alone files Finally we come to `\lstinputlisting`, the command 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[last=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. It is the effect of three parameters, which are explained later (keyword: 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 problem. 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{red}\bfseries\underbar,
                                % underlined bold red keywords
    identifierstyle=,           % nothing happens
    commentstyle=\color{white}, % white comments
    stringstyle=\ttfamily,      % typewriter type for strings
    stringspaces=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 and `\underbar` underlines the argument following up. 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 following tokens—or you will get deep in trouble.

→ 'basicstyle=\small' looks fine, but comments look really bad with 'commentstyle=\small' and empty basic style. The package adjusts internal data after selecting the basic style at the beginning of each listing. This can be a problem if you change the font size for comments or strings, for example. In this case you might want to use 'fontadjust=true' to update the internal data every font selection. Don't use this parameter otherwise!

Warning You should be very careful with striking styles; the last 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 also provides information.

Line numbers Apart from code fragments you can get numbered code lines, e.g. tiny numbers, each second line, with 5pt distance to the listing:

```
\lstset{labelstyle=\tiny, labelstep=2, labelsep=5pt}%            <===
```

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

6 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. 'labelstep=0' turns them off.

→ Can I use these parameters in the optional arguments? Of course. Note that optional arguments modify values for one particular listing only, i.e. you change the appearance, step or distance of line numbers for a single listing. The previous values are restored afterwards.

The `lstlisting` 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 listings in between. Read section 2.7 for a thorough discussion.

Floating listings Program listings except code fragments 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≤0) **then** i ←1;
if (i≥0) **then** i ←0;
if (i≠0) **then** i ←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

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 ∀ 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.

Visit the home page at www-inf.enst.fr/~demaille/a2ps.

cvt2ltx is a family of 'source code to LaTeX' converters for C, Objective C, C++, IDL and Perl. Other programming languages can be added, but currently it isn't documented how this is done.

Available via ftp from axp3.sr.fh-mannheim.de/cvt2latex.

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. Line numbers to the left or right, arbitrary LaTeX code in the source code, printing symbols instead of verbose names, font setup, and more is supported. You will (have to) like it (if you don't like listings).

Available via ftp from CTAN/support/lgrind.

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.

Available via ftp from CTAN/support/C++2LaTeX-1_1pl1.

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.

Available via ftp from CTAN/support/slatex.

tiny_c2ltx is a C/C++ to LaTeX converter based on cvt2ltx (or the other way round?). It supports block comments, LaTeX code in/as comments, and smart line breaking. The package does not provide line numbers. Font selection and tabulators are hard-coded, i.e. you have to rebuild the program if you want to change the appearance.

Available via ftp from CTAN/support/tiny_c2l.

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.

Available via ftp from CTAN/macros/latex/contrib/other/misc.

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

Available via ftp from CTAN/macros/latex/contrib/other/alg.

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.

Available via ftp from CTAN/macros/latex/contrib/supported/algorithms.

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.

This package is part of the LaTeX base distribution.

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

Available via ftp from CTAN/macros/latex/contrib/supported/moreverb.

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

Available via ftp from CTAN/macros/latex/required/tools.

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.

Available via ftp from CTAN/macros/latex/contrib/supported/fancyvrb.

→ Why do you list all these alternatives? Well, it's always good to know the competitors. And trying a different package and coming back is better than the other way round.

2 The next steps

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'. A 'driver' is generated from `lstdrvrs.dtx`.

Copyright The listings package is copyright 1996–2000 Carsten Heinz. The drivers are copyright 1997/1998/1999/2000 any individual author listed in the driver files.

Distribution and warranty The listings package as well as `lstdrvrs.dtx` and all drivers are distributed under the terms of the LaTeX Project Public License from CTAN archives in directory `macros/latex/base/lppl.txt`, either version 1.0 or any later version.

Use of the package The listings package is free software. However, if you distribute the package as part of a commercial product or if you use the package to prepare a commercial document (books, journals, and so on), I'd like to encourage you to make a donation to the LaTeX3 fund. The size of this 'license fee' should depend on the value of the package for your product. For more information about LaTeX3 see http://www.latex-project.org.

If you use the package to typeset a commercial or non-commercial document, please send me a copy of the document (`.dvi`, `.ps`, `.pdf`, hardcopy, etc.) to support further development.

Modification advice Permission is granted to modify the listings package as well as `lstdrvrs.dtx`. You are not allowed to distribute any changed version of the package or any changed version of `lstdrvrs.dtx`, neither under the same name nor under a different one. Instead contact the address below. Other users will welcome removed bugs, new features, and additional programming languages.

Contacts Send your comments, ideas, bug reports and additional programming languages to *Carsten Heinz, Tellweg 6, 42275 Wuppertal, Germany* or preferably to `cheinz@gmx.de` using `listings` in the subject.

Mailing list This is mainly an announcement list regarding new versions, bugs, patches, and work-arounds. So I recommend it for system administrators, maintainers of LaTeX installations, or people who absolutely need the latest bugs. To join the list, send an email to `cheinz@gmx.de` with subject `subscribe listings`.

2.2 Installation

Software installation

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

`listings.dvi`	→	`texmf/doc/latex/listings`
`listings.dtx`, `listings.ins`, `lstdrvrs.dtx`, `lstpatch.sty`	→	`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. This bundle is available via ftp from CTAN/macros/latex/required/graphics.

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 4.15), 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 language definitions and add-ons (= aspects).

→ What does the label "data" mean? It indicates that the definition isn't a usual command. If you want to adjust this parameter, you have to redefine it via '`\renewcommand`'. For example, after '`\renewcommand\lstaspectfiles{}`' the package won't find required add-ons. Note that such redefinitions must not take any arguments!

data `\lstlistlistingname` contains `Listings`

The header name for the list of listings.

data `\lstlistingname` contains `Listing`

It's the string used to label the caption.

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

defines ⟨*dialect*⟩ as default dialect for ⟨*language*⟩. This dialect will be used for ⟨*language*⟩ if no dialect is given explicitly. Table 1 shows all predefined languages and dialects.

→ '`defaultdialect`' isn't a command. How can I use it? Remember the parameters in section 1.3: take the name plus equality sign plus value and use this as argument to '`\lstset`'. If you separate such "key=value"s by commas, you can set two or more default dialects with a single command. The standard configuration file '`listings.cfg`' serves as example.

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

defines an alias for a programming language. Each ⟨*alias*⟩ dialect 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`.

2.3 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 option loads an additional listings aspect (collection of commands and parameters). Usually you don't have to take care of such aspect loading. 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 very special features.

`0.19`

to compile documents created with version 0.19. This should be fully compatible except that the command `\lststorekeywords` doesn't exist. In fact, one should write "This is going to be ... in some later version" since I haven't cared much about the compatibility mode.

`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.

`procnames`

defines the keys of this experimental aspect, see 5.2.

`hyper`

defines keys for hyper referencing with hyperref, see 5.3.

`lgrind`

defines the `lgrindef` key, see 5.5.

After package loading I 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*⟩`}`

loads all specified languages. Each programming language is given in the form `[`⟨*dialect*⟩`]`⟨*language*⟩. Without the optional `[`⟨*dialect*⟩`]` the package loads a default dialect. So write '`[Visual]C++`' if you want Visual C++ and '`[ANSI]C++`' for ANSI C++.

After or even before language loading, you might want to define default dialects—just to be independent of configurations files.

2.4 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, `first` 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: `first=2,last=5`.

→ So I can write '`\lstset{first=2,last=5}`' once for all? No. '`first`' and '`last`' 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.

2.5 Languages and styles

You already know that the key `language` activates programming languages—at least Pascal. The key provides an optional parameter to select particular dialects = versions or implementations of a language: `language=[`⟨*dialect*⟩`]`⟨*language*⟩. `language=[77]Fortran` selects Fortran 77 and `language=[XSC]Pascal` does the same for Pascal XSC. Table 1 shows all predefined languages and dialects. Use the names as ⟨*language*⟩ and ⟨*dialect*⟩, respectively. After `language={}` as argument to `\lstset` or as optional argument, no keywords are detected, no comments, no strings, and so on.

Each underlined dialect in the table is default dialect; it is selected if you leave out the optional argument. But note that predefined default dialects might change: it's either a standard dialect or the newest version. Moreover, a local configuration file can also change settings. Thus: If you make use of default dialects, define them in your document.

→ 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.

→ 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.

It's obvious that a pretty-printing tool requires some kind of language selection and definition. And it is very convenient to have the same for printing

Table 1: Predefined languages

Ada (83,95)	Algol (60,68)
C (ANSI,Objective)	C++ (ANSI,Visual)
Caml (light,Objective)	Clean
Cobol (1974,1985,ibm)	Comal 80
csh	Delphi
Eiffel	Elan
Euphoria	Fortran (77,90,95)
Haskell	HTML
IDL (empty,CORBA)	Java
Lisp	Logo
make (empty,gnu)	Mathematica (1.0,3.0)
Matlab	Mercury
Miranda	ML
Modula-2	Oberon-2
OCL (decorative,OMG)	Pascal (Borland6,Standard,XSC)
Perl	PL/I
POV	Prolog
Python	SHELXL
Simula (67,CII,DEC,IBM)	SQL
TeX (AlLaTeX,common,LaTeX,plain,primitive)	
VBScript	VHDL

styles: `\lstdefinestyle{`⟨*style name*⟩`}{`⟨*key=value list*⟩`}` stores a key=value list and the key `style=`⟨*style name*⟩ activates it. For example, you could write

```
\lstdefinestyle{number}
    {labelstep=1, labelstyle=\tiny, labelsep=10pt}
\lstdefinestyle{nonumber}
    {labelstep=0}
```

and switch from listings with line numbers (`style=number`) to listings without ones (`style=nonumber`). The advantage: styles at a central place of your document can be modified easily and the changes take effect on all listings.

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

2.6 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}
```

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. If you want to change settings for a single listing, use the optional argument.

Read also the paragraph about visible tabulators below.

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 (of code 128–255) directly and use these characters 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 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 *3.1 Troubleshooting*.

The extended characters don't cover Arabic, Chinese, Hebrew, Japanese, and so on. Read section 3.2 for details on work-arounds.

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.

Visible tabulators and spaces The verbatim part of the last example shows all spaces explicitly. This is also possible with tabulators.

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

```
\lstset{visiblespaces=true, % <===
        visibletabs=true,   % <===
        tab=\rightarrowfill}% <===
\begin{lstlisting}{}
    for i:=maxint to 0 do
    begin
    { do nothing }
    end;
\end{lstlisting}
```

If you request `visiblespaces` but no `visibletabs`, 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!

2.7 Line numbers

You already know the keys `labelstyle`, `labelstep`, and `labelsep` from section 1.4. Here now we begin with continued listings. Remember that the `lstlisting` environment has a name argument. Listings with identical names (case sensitive!) have a common line counter.

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

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

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

And we continue the listing:

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

The next `Test` listing goes on with line number 8, no matter whether there are other listings in between. Note that the empty line at the end of the first part is not printed here, but it counts for line numbering. The continue mechanism has two exceptions: an empty named (= `{}`) listing always starts with line number 1, a space named (= `{ }`) listing continues the last empty or space named one.

In fact, that's not true. The key `firstlabel` controls the line number of the first printed line:

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

And we continue the listing:

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

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

\end{lstlisting}
And we continue the listing:
\begin{lstlisting}[firstlabel=1]{ }
Write('Case insensitive ');
WritE('Pascal keywords.');
\end{lstlisting}
```

→ Okay. And how can I get decreasing line numbers? Sorry, what? Decreasing line numbers as on page 34. May I suggest to demonstrate your individuality by other means? If you differ, you should try a negative 'labelstep' (together with 'firstlabel').

Read section 3.5 on how to reference line numbers.

2.8 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`.

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}`. The optional argument of `caption` can be used to specify a short caption for the list of listings. If this short caption is empty then the listing will neither appear in that list nor it gets a number. But hold on. You've got to be aware that the key is used on an individual listing. Therefore you can't type `caption=[short]long` since the right bracket after `short` ends the optional argument of the pretty-printing command. It works if you enclose the whole value in braces: `caption={[short]long}`. By the way: the list of listings is printed via `\lstlistoflistings`.

If you want to drop the label `Listing` and the number, you should use `title`:

```
\begin{lstlisting}[frame=tb,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;
```

Frames The main key for frames is `frame`. If you use any subset of `trbl` as value, you get rules at the top, right, bottom, and/or left. Upper case letters will draw 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}
```

→ 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 4.5 heavily. And do this step by step! For example, begin with '`framerulewidth=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.

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`.

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

```
\definecolor{lightgray}{rgb}{0.75,0.75,0.75}
\lstset{backgroundcolor=lightgray}
```

```
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}
```

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

→ I want coloured space around the whole listing and can't get it, even not with the keys described in section 2.10. Try frames with framerule and background colour being equal.

2.9 Emphasize identifiers

Recall the pretty-printing commands and environment. `\lstinline` prints code fragments, `\lstinputlisting` whole files, and `lstlisting` prints 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. 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 if 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 function names `square` and `root`, for example, by underlining them. Then you could do it like this:

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

```
\lstset{emph={square,root},
        emphstyle=\underbar}
\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. And before you ask your next question, here is the answer: Yes, there is more than one `emph` 'class' and each class has its own style.

Both keys have an optional ⟨*class number*⟩ argument. Please note again: If you use a list of identifiers or if you use an optional argument of a key inside an optional argument of a pretty-printing command, you *must* put braces around the value. Though it is not necessary, the following example uses these braces. They are typically forgotten when they become necessary, after copy&paste or extending a list of identifiers, for example.

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

2.10 Listing alignment

The examples are typeset with centered `minipage`s. That's the reason why you can't see that line numbers are printed in the margin. Now we separate the minipage margin and the minipage by a vertical rule:

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

Some text before

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

The listing is lined up with the normal text. The parameter `indent` moves the listing to the right (or left if the dimension is negative).

```
Some text before
\begin{lstlisting}[indent=15pt]{}
for i:=maxint to 0 do
begin
    { do nothing }
end;
\end{lstlisting}
```

Some text before

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

```
\begin{lstlisting}{ }
Write('Insensitive');
WritE('keywords.');
\end{lstlisting}
```

```
Write('Insensitive');
6 WritE('keywords.');
```

Note again that optional arguments change settings for single listings.

If you use environments like `itemize` or `enumerate`, there is 'natural' indention coming from these environments. By default the listings package respects this. But you might use `wholeline=true` (or `false`) to make your own decision. You can use it together with `indent`, of course.

→ I get heavy overfull \hboxes from some listings. This comes from long lines in your listings. You have some options to get rid of the overful \hboxes. Firstly I recommend to typeset listings in smaller fonts than the surrounding text, for example 'basicstyle=\small'. Secondly you might want to use the flexible column format. Thirdly you can 'spread' the line width or set it explicitly, refer section 4.9. If all this doesn't help, you might want to change 'basewidth', but be careful! The two unknown items are explained in the next section.

You might need to control the vertical position of listings with the `boxpos` key, for example, if you use them in `minipage` or `tabular` environments. Here 'listings' means `lstlisting` or `\lstinputlisting`. As the following example shows, you can even place such listings inside paragraphs, but you must force the package to do this by enclosing the listing in `\hbox{` and `}`.

→ Is it good form to use the TEX-primitive '\hbox' in a LATEX document? No, it's not. But LATEX's '\mbox' does not work in this example:

```
Here are some multi-line listings inside a paragraph.
The 'boxpos' key controls their vertical alignment:
\hbox{\begin{lstlisting}[boxpos=c]{}          % <===
center
center
\end{lstlisting}}
\hbox{\begin{lstlisting}[boxpos=b]{}          % <===
bottom baseline
bottom baseline
\end{lstlisting}}
\hbox{\begin{lstlisting}[boxpos=t]{}          % <===
top baseline
top baseline
\end{lstlisting}}
```

Here are some multi-line listings inside a paragraph. The 'boxpos' key controls their vertical alignment: center center bottom baseline bottom baseline top baseline top baseline

2.11 Fixed and flexible columns

The first thing a reader notices—except different styles for keywords, etc.—is bad column alignment like this:

```
if x=y then write('align')
      else print('align');
```

This is not an illustration of the flexible column format. But the piece of code has been typeset with the listings package using some unusual settings. However, the column alignment can't be disturbed if we put the characters in boxes of identical width:

```
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 ' ) ;
```

The default procedure of this package works with a slight modification. All input will be cut up in units to find keywords. We put each unit in a box, which width is multiplied by the number of characters we put in, of course. The result is

```
if  x=y  then   write (' align ')
         else    print (' align ');
```

Since we put wide and thin characters in the same box, the width of a single character box need not to be the width of the widest character. The empirical value 0.6em (which is called 'base width' later) is a compromise between overlapping characters and the number of boxes not exceeding the line width, i.e. how many characters fit a line without getting an overfull `\hbox`.

But overlapping characters are a problem if you use many upper case letters, e.g. WOMEN—blame me and not the women, in fact MEN doesn't look better. The flexible column format typesets all characters at their natural width. In particular characters never overlap. If a word requires more space than reserved, the rest of the line simply moves to the right. If a following word needs less space than reserved or if there are spaces following each other, this space is used to fix the column alignment. Arne John Glenstrup (whose idea the format was) pointed out that he had good experience with flexible columns and assembler listings. The differences can be summed up as follows: The fixed column format ruins the nice spacing intended by the font designer, and the flexible format ruins the column alignment (possibly) intended by the programmer. We illustrate that.

verbatim	fixed columns with 0.6em	flexible columns with 0.45em
`WOMEN  are`	WOMEN are	WOMEN are
`MEN`	MEN	MEN
`WOMEN are`	WOMEN are	WOMEN are
`better MEN`	better MEN	better MEN

Hope this helps. In flexible mode, one of the two blanks in the first line is used to fix the column alignment. This is unlike TEX's glue: the first 'crumple zones' take it all. There is never a crumble zone if 0em is the base width. In this case the first 'MEN' would had to the left since the preceding spaces were $7 \cdot 0\text{em} = 0\text{em}$ wide. If you use such extreme values, you should try `keepspaces=true` to protect the spaces.

→ 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" ...

`flexiblecolumns=`⟨`true`|`false`⟩ turns the flexible columns on and off, respectively. The predefinition of the 'base width' is `basewidth={0.6em,0.45em}`, where the first value is for fixed mode and the second for flexible columns. Change it if you like, but be very careful!

2.12 Indexing

is just like emphasizing identifiers—I mean the usage:

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

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

→ 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 (CTAN/macros/latex/contrib/supported/camel). 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]`'.

2.13 Closing and credits

You've seen a lot of keys but you are far away from knowing all of them. The next step would be real use of the listings package. If you encounter any problems or need some special things, come back to this documentation. Look up the known commands and keys in the reference guide; then you should be able to understand and use all the other. Complain if this is not true: email to `cheinz@gmx.de`.

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

Andreas Bartelt, Jan Braun, Denis Girou, Arne John Glenstrup,
Rolf Niepraschk, Rui Oliveira and Boris Veytsman.

Moreover I wish to thank

Bjørn Ådlandsvik, Gaurav Aggarwal, Jason Alexander,
Donald Arseneau, Claus Atzenbeck, Peter Bartke,
Olaf Trygve Berglihn,Peter Biechele, Kai Below, David Carlisle,
Patrick Cousot, Holger Danielsson, Detlev Dröge,
Anders Edenbrandt, David John Evans, Harald Harders,
Christian Haul, Aidan Philip Heerdegen, Jim Hefferon, Jürgen Heim,
Dr. Jobst Hoffmann, Torben Hoffmann, Berthold Höllmann,
Ralf Imhäuser, R. Isernhagen, Marcin Kasperski, Dr. Peter Leibner,
Thomas Leduc, Magnus Lewis-Smith, Andreas Matthias,
Knut Müller, Torsten Neuer, Heiko Oberdiek, Zvezdan V. Petkovic,
Michael Piotrowski, Manfred Piringer, Vincent Poirriez, Ralf Quast,
Aslak Raanes, Detlef Reimers, Magne Rudshaug, Andreas Stephan,
Gregory Van Vooren, Dominique de Waleffe, Michael Weber,
Sonja Weidmann, Herbert Weinhandl, Michael Wiese, Jörn Wilms
and Kai Wollenweber.

This list is probably not complete since I have't updated it at all. I'll use the next release to do so.

3 Tips and tricks

3.1 Troubleshooting

Before you make a bug report, consult the reference guide whether the problem is already known. If not, please try to locate the problem. Start from the minimal in section 1.1. If you use other packages, load only the required ones. Then add the LaTeX code which causes the problem, but keep it short and eliminate packages not necessary. Remove some code from the file until the problem disappears. Then you've found a crucial piece. Start over with removing until all code is substantial. Then send a bug report via email to `cheinz@gmx.de` and include the now modified minimal file and 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.

3.2 National characters

Apart from typing in national characters directly, you can use the 'escape' feature described in section 4.11. The keys `escapechar`, `escapeinside`, and `texcl` allow partial usage of LaTeX code, for example:

```
\begin{lstlisting}[escapechar=‘]{}
‘\"a \‘e {\=\i} {\oe} \u u‘
\end{lstlisting}
```

ä è ī œ ŭ

The escape character delimits the LaTeX code: the first reverse apostrophe starts the escape, the second belongs to the grave accent command, and the third eventually ends the escape.

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 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!

3.3 Listings with 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 (if on) 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.

3.4 Bold typewriter fonts

Many people asked for bold typewriter fonts since they aren't included in the LaTeX standard distribution. Here now one answer on how to use them in spite of that. Firstly you'll need Metafont source files for bold typewriter, e.g. `cmbtt8.mf`, `cmbtt9.mf` and `cmbtt10.mf` from CTAN/fonts/cm/mf-extra/bold. Secondly you have to create `.tfm`-files, i.e. run the Metafont program on these sources. This is possibly done automatically when you use the fonts in a document. Finally you must tell LaTeX that you've installed bold typewriter fonts. Just use

```
\DeclareFontShape{OT1}{cmtt}{bx}{n}
     {<5><6><7><8>cmbtt8%
      <9>cmbtt9%
      <10><10.95>cmbtt10%
      <12><14.4><17.28><20.74><24.88>cmbtt10%
      }{}
```

(before `\begin{document}`). That's all!

3.5 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.

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

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

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

  Line 3 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.

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 Data types

General notes The parameters of commands and keys are specified either by their type or via explicitly given arguments. For example, a key is presented either as 'key=value' pair or as 'key=data type'. Both value and data type will be enclosed in ⟨ ⟩. Most data types and values are self-explanatory. However some hints can't be wrong.

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. If you use an optional argument of a key inside an optional key=value list, you must put braces around the whole value.
3. A vertical rule indicates an alternative, e.g. ⟨`true`|`false`⟩ allows `true` or `false` as arguments.
4. If you need one of the special characters `{}#%\` in or as an argument, the character(s) must be preceded by a backslash. This means that you must write `\}` for the single character 'right brace', for example.

Some data types

⟨*basic style*⟩—token sequence for type selection (size, typeface, colour, etc.)

⟨*character*⟩—a single character

⟨*character sequence*⟩—a character string

⟨*delimiter*⟩—a character string used as delimiter

⟨*dimension*⟩—a TEX dimension

⟨*identifiers*⟩—a list of identifiers

⟨*key=value list*⟩—a list of key=value pairs

⟨*keywords*⟩—a list of keywords

⟨*keyword classes*⟩—a list of keyword classes; keyword classes are, for example, `keywords`, `keywords2`, and `texcs`

⟨*number*⟩—a TEX number

⟨*style*⟩—like ⟨*basic style*⟩ but the very last token *might* take exactly one argument, namely the character string to typeset

⟨*subset of ...*⟩—any combination of the characters

⟨*tokens*⟩—arbitrary token sequence (potentially unsafe since arbitrary, so use it wisely)

Scheme of presentation

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

explanation and more details

The label in the left margin (if present) provides information about the command, environment or key: '*addon*' indicates additional functionality, '*new*' a new and '*changed*' a modified key, '*data*' a data containing command (which is therefore adjustable via `\renewcommand`), and so on.

The label in the right margin is the introductory version number. If you find verbatim text next to the number then this is the predefined value. Note that some keys are reset every listing, namely the keys which can be used on individual listings only.

4.2 Languages and styles

Table 1 on page 14 shows all languages and dialects provided by `lstdrvrs.dtx`. They have all bugs coming from the language defining commands described in section 4.15, e.g. in Ada and Matlab it is still possible that the package assumes a string where none exists.

→ Err, have you just said thet the package isn't suitable to typeset Ada or Matlab code? No, sometimes the highlighting isn't correct. These rare cases are defined in section 4.15 in the paragraph about strings.

The 'empty' language detects no keywords, no comments, no strings, and so on. Note that the arguments ⟨*language*⟩, ⟨*dialect*⟩, and ⟨*style name*⟩ are case insensitive and that spaces have no effect.

`language=[`⟨*dialect*⟩`]`⟨*language*⟩ `{}` 0.17

activates a (dialect of a) programming language.

`defaultdialect=[`⟨*dialect*⟩`]`⟨*language*⟩ 0.19

defines ⟨*dialect*⟩ as default dialect for ⟨*language*⟩. This dialect will be used for ⟨*language*⟩ if no dialect is given explicitly. 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`.

`style=`⟨*style name*⟩ `{}` 0.18

activates the key=value list stored with `\lstdefinestyle`.

`\lstdefinestyle{`⟨*style name*⟩`}{`⟨*key=value list*⟩`}` 0.19

stores the key=value list.

→ It's easy to crash the package with 'style'. 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.

4.3 Typesetting listings

Please note that all optional ⟨*key=value list*⟩s modify parameters for single listings only.

`\lstset{`⟨*key=value list*⟩`}` 0.19

sets the values of the specified keys, see also section 2.4.

`\lstinline[`⟨*key=value list*⟩`]` 0.18

works like `\verb` but uses the active language and style. You can write '`\lstinline!var i:integer;!`' and get '**var** i:**integer**;'. Note that these listings use flexible columns except `flexiblecolumns=false` is a key=value pair in the optional argument.

`\lstinputlisting[`⟨*key=value list*⟩`]{`⟨*file name*⟩`}` 0.1

typesets the stand alone source code file as a displayed listing, i.e. the command starts a new paragraph for the listing.

`lstlisting[`⟨*key=value list*⟩`]{`⟨*name*⟩`}` 0.15

typesets the code between `\begin{lstlisting}` (+ arguments + line break) and `\end{lstlisting}` as a displayed listing. Source code directly before and LaTeX code after the end of environment is typeset respectively executed.

Same named listings have common line counters, i.e. the second (same named) listing continues the first, the third continues the second, and so on. There are two exceptions: An empty-named listing starts with line number 1 and is continued with space-named listings (= `{ }`).

`extendedchars=`⟨`true`|`false`⟩ or `extendedchars` `false` 0.18

allows or prohibits extended characters in listings, i.e. characters with codes 128–255. If you use extended characters, you should use the fontenc or inputenc package.

changed `gobble=`⟨*number*⟩ `0` 0.19

gobbles ⟨*number*⟩ characters at the beginning of each *environment* code line. Tabulators might expand to `tabsize` spaces before they are gobbled. Code lines with less than ⟨*number*⟩ characters are viewed empty.

Don't indent the end of environment by more than `gobble` characters, but less characters are allowed.

`first=`⟨*number*⟩ `1` 0.1

`last=`⟨*number*⟩ `9999999` 0.1

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

`print=`⟨`true`|`false`⟩ or `print` `true` 0.12

controls whether displayed listings are typeset. If you use `print=false` at the beginning of a document to compile a draft version, you might use `print` in optional arguments to typeset particular listings despite of that.

`showlines`=⟨`true`|`false`⟩ or `showlines` `false` 0.20

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

`float`=⟨*subset of* `tbph`⟩ or `float` `tbp` 0.20

makes sense with 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.

`boxpos`=⟨`b`|`c`|`t`⟩ `c` 0.18

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.

Note that `\hbox`ed listings don't use `spread`, for example.

new `aboveskip`=⟨*dimension*⟩ 0.21

new `belowskip`=⟨*dimension*⟩ 0.21

define the space above and below displayed listings.

`lineskip`=⟨*dimension*⟩ `0pt` 0.17

specifies the additional space between lines in listings.

4.4 Figure out the appearance

`basicstyle`=⟨*basic style*⟩ `{}` 0.18

`identifierstyle`=⟨*style*⟩ `{}` 0.18

`commentstyle`=⟨*style*⟩ `\itshape` 0.11

`stringstyle`=⟨*style*⟩ `{}` 0.12

`keywordstyle`=⟨*style*⟩ `\bfseries` 0.11

`ndkeywordstyle`=⟨*style*⟩ `keywordstyle` 0.19

optional `texcsstyle`=⟨*style*⟩ `keywordstyle` 0.20

optional `directivestyle`=⟨*style*⟩ `keywordstyle` 0.20

determine the style in which special parts of a listing appear. The *last* token (except `basicstyle`) might be an one-parameter command like `\textbf` or `\underbar`.

new `emph=[`⟨*number*⟩`]{`⟨*identifiers*⟩`}` 0.21

new `moreemph=[`⟨*number*⟩`]{`⟨*identifiers*⟩`}` 0.21

new `deleteemph=[`⟨*number*⟩`]{`⟨*identifiers*⟩`}` 0.21

define, add and remove ⟨*identifiers*⟩ from 'emphasize class ⟨*number*⟩'. If you don't give an optional argument, the package assumes ⟨*number*⟩ = 1.

new `emphstyle=[`⟨*number*⟩`]{`⟨*style*⟩`}` 0.21

defines the style for class ⟨*number*⟩.

`stringspaces=`⟨`true`|`false`⟩ `true` 0.12

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

`visiblespaces=`⟨`true`|`false`⟩ `false` 0.20

lets all blank spaces appear ␣ or as blank spaces.

`visibletabs=`⟨`true`|`false`⟩ `false` 0.20

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*⟩ 0.20

⟨*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.

`tabsize=`⟨*number*⟩ `8` 0.12

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.

`formfeed=`⟨*tokens*⟩ `\bigbreak` 0.19

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

4.5 Frames

`frame=`⟨*subset of* `trblTRBL`⟩ `{}` 0.19

The characters `trblTRBL` are attached to lines at the top and bottom of a listing and to lines on the right and left. There are two lines if you use upper case letters. If you want a single frame around a listing, write `frame=tlrb` or `frame=bltr`, for example. If you want double lines at the top and on the left and no other lines, write `frame=TL`.

Note that frames reside outside the listing's space. Use `spread` if you want to shrink frames (to `\linewidth` for example) and use `indent` to move line numbers inside frames.

`framerulewidth=`⟨*dimension*⟩ `0.4pt` 0.19

`framerulesep=`⟨*dimension*⟩ `2pt` 0.19

These keys control the width of the rules and the space between double rules.

`frametextsep=`⟨*dimension*⟩ `3pt` 0.19

controls the space between frame and listing.

`framespread=`⟨*dimension*⟩ `0pt` 0.20

makes the frame on each side half ⟨*dimension*⟩ wider.

`frameround=`⟨`t`|`f`⟩⟨`t`|`f`⟩⟨`t`|`f`⟩⟨`t`|`f`⟩ `ffff` 0.20

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 `frametextsep` and is independent from `framespread`. The size is possibly adjusted to fit LaTeX's circle sizes.

new `backgroundcolor={`⟨*color model*⟩`}`⟨*color*⟩ 0.21

new `framerulecolor={`⟨*color model*⟩`}`⟨*color*⟩ 0.21

specify the colour of the background and the rules respectively. Note that you need the color package to use these keys.

`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 report in.

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

```
\lstset{framespread=5mm}
\begin{lstlisting}[frame=trbl]{}
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*⟩`}` 0.20

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.

4.6 Captions

In despite of LaTeX standard behaviour captions and floats are independent from each other here. You can use captions with non-floating listings. It's your choice whether a titled listing also gets a number, how the number looks like, and so on.

`title=`⟨*title text*⟩ 0.21

can be used on individual displayed listings only. ⟨*title text*⟩ is used for a title without any numbering and without a header.

`caption={[`⟨*short*⟩`]`⟨*caption text*⟩`}` 0.20

can be used on individual displayed listings only. If you don't use `[`⟨*short*⟩`]`, the package assumes ⟨*short*⟩=⟨*caption text*⟩. If ⟨*short*⟩ is empty, the listing is neither numbered nor it appears in the list of listings.

Note: The braces around the value are necessary if and only if you use the optional ⟨*short*⟩ argument (or if ⟨*caption text*⟩ contains `]`).

new `label=`⟨*name*⟩ 0.21

makes a listing with non-empty ⟨*short*⟩ referable via `\ref{`⟨*name*⟩`}`.

`\lstlistoflistings` 0.16

prints a list of listings. The names are the (short) captions, file names or names of the listings.

data `\lstlistlistingname` `Listings` 0.16

The header name for the list of listings.

data `\lstlistingname` `Listing` 0.20

The header name for listings with captions.

data `\thelstlisting` `\arabic{lstlisting}` 0.20

prints the caption's label number.

`captionpos=`⟨*subset of* `tb`⟩ `t` 0.20

specifies the position(s) of the caption.

`abovecaptionskip=`⟨*dimension*⟩ `\smallskipamount` 0.20

`belowcaptionskip=`⟨*dimension*⟩ `\smallskipamount` 0.20

is the vertical space above respectively below each caption.

4.7 Labels

`labelstep=`⟨*number*⟩ `0` 0.16

All lines with "line number ≡ 0 modulo ⟨*number*⟩" get a label. Usually this label is the line number, but it's controlled by `labelstyle` and `\thelstlabel`. ⟨*number*⟩ = 0 turns the labels off.

`labelstyle=`⟨*style*⟩ `{}` 0.16

determines the font and size of the labels.

data `\thelstlabel` `\arabic{lstlabel}` 0.20

prints the lines' label numbers.

`labelsep=`⟨*dimension*⟩ `10pt` 0.19

is the distance between label and listing.

`firstlabel=`⟨*number*⟩ 0.20

`advancelabel=`⟨*number*⟩ `0` 0.19

sets respectively advances the number of the first label. Both keys must be used in the optional key=value list.

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

```
\renewcommand*\thelstlabel{\oldstylenums{\the\value{lstlabel}}}
```

```
\begin{lstlisting}[firstlabel=753]{}
begin { empty lines }
```

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

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

Exercise: 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. Minimal means nonnegative number here. 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!

4.8 Indexing

addon `index=[`⟨*number*⟩`][`⟨*keyword classes*⟩`]{`⟨*identifiers*⟩`}` 0.19

new `moreindex=[`⟨*number*⟩`][`⟨*keyword classes*⟩`]{`⟨*identifiers*⟩`}` 0.21

new `deleteindex=[`⟨*number*⟩`][`⟨*keyword classes*⟩`]{`⟨*identifiers*⟩`}` 0.21

define, add and remove ⟨*identifiers*⟩ and ⟨*keyword classes*⟩ from index list no. ⟨*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.

renamed,addon `indexstyle=[`⟨*number*⟩`]`⟨*tokens* (*'one parameter' macro*)⟩ `\lstindexmacro` 0.19

⟨*tokens*⟩ actually indexes the identifiers for list no. ⟨*number*⟩. In contrast to the style keys, ⟨*tokens*⟩ *must* read exactly one parameter, namely the identifier. Default definition is

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

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

4.9 Line shape and breaking

`linewidth=`⟨*dimension*⟩ `\linewidth` 0.21

defines the base line width for listings. Please note that other keys, e.g. `spread`, are taken into account additionally.

bug `spread=`⟨*dimension*⟩ or `spread={`⟨*inner*⟩`,`⟨*outer*⟩`}` `0pt` 0.16

defines *additional* line width for listings, which may avoid overfull `\hbox`es if a listing has long lines. The inner and outer spread is given explicitly or is equally shared. For one sided documents 'inner' and 'outer' have the effect of 'left' and 'right'. Note that `indent` is always 'left'.

Bug (two sided documents): At top of page it's possible that the package uses inner instead of outer spread or vice versa. This happens when TEX finally moves one or two source code lines to the next page, but hasn't decided it when the listings package processes them. Work-around: interrupt the listing and/or use an explicit `\newpage`.

`indent=`⟨*dimension*⟩ `0pt` 0.19

indents each listing by ⟨*dimension*⟩. This is the best way to move line numbers and the listing to the right (or left if the dimension is negative).

`wholeline=`⟨`true`|`false`⟩ `false` 0.19

prevents or lets the package use indention from list environments like `enumerate` or `itemize`.

`breaklines=`⟨`true`|`false`⟩ or `breaklines` `false` 0.20

activates or deactivates automatic line breaking of long lines.

`breakindent=`⟨*dimension*⟩ `20pt` 0.20

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

`breakautoindent=`⟨`true`|`false`⟩ or `breakautoindent` `true` 0.20

activates or deactivates automatic indention of broken lines. This indention is used additionally to `breakindent` and is equal to the indention of the source code line, see the example below.

`visiblespaces=true` converts 'invisibles' spaces and tabulators to visible ␣. This will set 'auto indent' to 0pt, i.e. there is no automatic indention.

`prebreak=`⟨*tokens*⟩ `{}` 0.20

`postbreak=`⟨*tokens*⟩ `{}` 0.20

⟨*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*⟩.

We use tabulators now to create long lines, but the verbatim part uses `tabsize=1`.

```
\lstset{postbreak=\space\space,breakindent=20pt,breaklines}

\begin{lstlisting}{}
  "A very long string doesn't fit the current line width."
    "An even longer line doesn't fit also, of course, and goes over three lines."
\end{lstlisting}

\begin{lstlisting}[breakautoindent=false]{}
  { Now auto indention is off, and only breakindent=20pt and postbreak are used. }
\end{lstlisting}

\begin{lstlisting}[visiblespaces]{}
  { 'visiblespaces=true' implies 'breakautoindent=false'. }
\end{lstlisting}
```

```
"A␣very␣long␣string␣doesn't␣fit␣the␣current␣line␣width."
                    "An␣even␣longer␣line␣doesn't␣fit␣also,␣of␣course,␣
                        and␣goes␣over␣three␣lines."

              { Now␣auto␣indention␣is␣off,␣and␣only␣breakindent=20pt␣and␣postbreak
  are␣used.␣}

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{␣'␣visiblespaces␣=true'␣implies␣'breakautoindent=false'.␣}
```

4.10 Column alignment

`flexiblecolumns=`⟨`true`|`false`⟩ or `flexiblecolumns` `false` 0.18

selects the flexible respectively fixed column format, refer section 2.11.

`basewidth=`⟨*dimension*⟩ or 0.16

`basewidth={`⟨*fixed*⟩`,`⟨*flexible mode*⟩`}` `{0.6em,0.45em}` 0.18

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

`keepspaces=`⟨`true`|`false`⟩ `false` 0.21

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

`outputpos=`⟨`c`|`l`|`r`⟩ `c` 0.19

controls 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 respectively |listing|, |listing|, and | listing| with flexible columns.

`fontadjust=`⟨`true`|`false`⟩ or `fontadjust` `false` 0.20

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 4.4 make heavy font size changes, see the example below.

```
{ 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}
```

Note that `fontadjust` also effects the keywords!

4.11 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` 0.18

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` 0.19

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={}` `{}` 0.19

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={}` `{}` 0.20

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

`escapebegin=`⟨*tokens*⟩ `{}` 0.20

`escapeend=`⟨*tokens*⟩ `{}` 0.20

The tokens are executed at the beginning respectively at the end of each escape, in particular for `texcl`. See section 3.2 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}
```

```
// calculate a_{ij}
  a_{ij} = a_{jj}/a_{ij};
```

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

```
// calculate 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 in comment style and by the listings package. 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' without comment style.

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.

4.12 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 ; –).

bug `fancyvrb=`⟨`true`|`false`⟩ 0.19

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.

Bug (`commandchars`): If you use fancyvrb's `commandchars`, the used commands must not take arguments from the verbatim code except the source code which is actually typeset. For example, `\textcolor{red}{keyword}` is illegal since `red` is (used to select the colour and) not typeset. There is an easy work-around: write `\newcommand*\myred{\textcolor{red}}` and use `\myred{keyword}` inside the verbatim code.

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

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

\par\vspace{72.27pt}

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

```
First verbatim line.
Second verbatim line.
```

```
First verbatim line.
Second verbatim line.
```

The lines typeset by the listings package are wider since the default `basewidth` equals not the width of a single typewriter type character.

4.13 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 of parameters*⟩`][`⟨*opt. default arg.*⟩`]` 0.19
`{`⟨*starting code*⟩`}{`⟨*ending code*⟩`}`

Both `lstlisting` and version 0.17 `listing` environment are defined with this command. The latter one is quite simple since the one and only and optional argument is the name.

```
\lstnewenvironment{listing}[1][]
    {\gdef\lst@intname{#1}}
    {}
```

The other is more difficult. First we test whether the nonoptional name argument is an EOL character. If this is the case, the user has forgotten the name. Then we use the optional key=value list. The rest ensures correct (continued) line numbering.

```
\lstnewenvironment{lstlisting}[2][]
    {\lst@TestEOLChar{#2}%
     \lstset{#1}%
     \csname lst@SetFirstLabel\endcsname}
    {\csname lst@SaveFirstLabel\endcsname}
```

Finally note that all `lst`-environments can also be used in command fashion like this

```
\lstlisting[gobble=4]{}
    \begin{lstlisting}{}
    Silly sentence?
    \end{listings}
\endlstlisting
```

```
\begin{lstlisting}{}
Silly sentence?
\end{listings}
```

4.14 Language specific keys

optional `printpod=⟨true|false⟩` `false` 0.19

prints or drops PODs in Perl.

optional `usekeywordsinside=⟨true|false⟩` `true` 0.20

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

optional `makemacrouse=⟨true|false⟩` `true` 0.20

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)).

4.15 Language definitions

Language definitions and also some style definitions tend to have long definition parts. This is why I and possibly other people tend to forget commas between the key=value elements. If you select a language and get a `Missing = inserted for \ifnum` error, this is surely due to a missing comma after `keywords`=value. If you encounter unexspected characters after selecting a language (or style), you have either forgotten a comma or you have given to many arguments to a key, for example, `commentline={--}{!}`.

`\lstdefinelanguage` 0.19

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

defines a programming language. If the language definition is based on another, you must specify the whole `[`⟨*base dialect*⟩`]{`⟨*and base language*⟩`}`. An empty ⟨*base dialect*⟩ uses the default dialect! Selecting the new language executes the ⟨*key=value list*⟩ after selecting the base language.

The last optional argument should specify all required `lst`-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` with same syntax defines languages in the driver files.

`\lstalias{`⟨*alias*⟩`}{`⟨*language*⟩`}` 0.18

defines an alias for a programming language. Each ⟨*alias*⟩ dialect 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*⟩`}` 0.18

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`.

Note that a (local) configuration file possibly defines some aliases.

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!*

`keywords={`⟨*keywords*⟩`}` 0.11

`morekeywords={`⟨*keywords*⟩`}` 0.11

`deletekeywords={`⟨*keywords*⟩`}` 0.18

`ndkeywords={`⟨*keywords*⟩`}` 0.19

`morendkeywords={`⟨*keywords*⟩`}` 0.19

`deletendkeywords={`⟨*keywords*⟩`}` 0.19

define, add or remove the keywords from appropriate list. Please note the key `specialscan` below (if you don't use unusual charaters in keywords.)

optional `texcs={`⟨*list of control sequences (without backslashes)*⟩`}` 0.19

optional `moretexcs={`⟨*list of control sequences (without backslashes)*⟩`}` 0.20

defines or adds control sequences for TEX and LATEX.

optional `directives={`⟨*list of compiler directives*⟩`}` 0.18

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

optional `keywordsinside=`⟨*character*⟩⟨*character*⟩ or `keywordsinside={}` 0.20

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

`sensitive=`⟨`true`|`false`⟩ 0.14

makes the keywords case sensitive and insensitive, respectively. This key affect the keywords only in the phase of typesetting. In all other situations keywords are case sensitive, for example, `deletekeywords={save,Test}` removes 'save' and 'Test', but neither 'SavE' nor 'test'.

new `specialscan=`⟨`true`|`false`⟩ `true` 0.21

enables or disables (faster) the automatic scan for special characters. If deactivated, you must specify all special characters in the keywords with the following key(s):

`alsoletters={`⟨*character sequence*⟩`}` 0.19

`alsodigits={`⟨*character sequence*⟩`}` 0.19

`alsoother={`⟨*character sequence*⟩`}` 0.19

These keys support the 'special character' auto-detection of the keyword commands. For our purpose here, identifiers are out of letters (`A`–`Z`,`a`–`z`,`_`,`@`,`$`) and digits (`0`–`9`), but an identifier must begin with a letter. If you write `keywords={one-two,\#include}`, the minus becomes necessarily a digit and the sharp a letter since the keywords can't be detected otherwise. This means that the defined keywords affect the process of building the 'output units'!

The three keys overwrite such default behaviour. Each character of the sequence becomes a letter, digit and other, respectively.

`otherkeywords={`⟨*keywords*⟩`}` 0.20

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.

Strings Just two keys.

`stringtest=`⟨`true`|`false`⟩ 0.19

enables or disables string tests. If activated, line exceeding strings issue warnings and the package exits string mode.

`string=[`⟨`b`|`d`|`m`|`bd`⟩`]{`⟨*character sequence*⟩`}` 0.12

Each character might start a string or character literal. 'Stringizers' match each other, i.e. starting and ending delimiters are the same. The optional argument controls how the stringzier(s) itself is/are represented in a string or character literal: it is preceded 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 stringizers are also used for other purposes. In general the stringizer is also doubled, but a string does not start after a letter or a right parenthesis.

Comments If you have already defined any of the following comments and you want to remove it, let all arguments to the key empty.

`comment=[`⟨*type*⟩`][`⟨*type option*⟩`]`⟨*delimiter(s)*⟩ 0.13

`comment=[l]`⟨*delimiter*⟩ 0.13

`comment=[f][`⟨*n=preceding columns*⟩`]`⟨*character sequence*⟩ 0.18

`comment=[s]{`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}` 0.13

`comment=[d]{`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}` 0.13

`comment=[n]{`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}` 0.13

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

Each given character becomes a 'fixed comment line' separator: it starts a comment line if and only if it is in column $n + 1$. Fortran 77 declares its comments via `fixedcommentline=*Cc` ($n = 0$ is default).

Here we have two or four delimiters. The second ends a comment starting with the first, and similarly the fourth and third delimiter for double comments. If you require three such comments you can use `singlecomment` and `doublecomment` at the same time. C, Java, PL/I, Prolog and SQL all define single comments via `singlecomment={/*}{*/}`, and Algol does it with `singlecomment={\#}{\#}`, which means that the sharp delimits both beginning and end of a single comment.

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

optional `keywordcomment={`⟨*keywords*⟩`}` 0.17

optional `keywordcommentsemicolon={`⟨*keywords*⟩`}{`⟨*keywords*⟩`}{`⟨*keywords*⟩`}` 0.17

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

Defining a (double) keyword comment semicolon needs 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`⟩ 0.17

activates or deactivates PODs—Perl specific.

5 Experimental features

This section describes the more or less unestablished parts of this package. It's unlikely that they are removed (except it is stated explicitly), but they are liable to (heavy) changes and improvements.

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, the 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. Now environment examples:

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

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

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

```
We need no protection here,
 but  in  this  line .
```

```
\fbox{%^^J
\begin{lstlisting}{}^^J
We need no protection here,^^J
\ but\ \ in\ \ this\ \ line.^^J
\end{lstlisting}}
```

→ You might wonder that this feature is still experimental. The reason: You shouldn't use listings inside arguments.

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.

optional `prockeywords={`⟨*keywords*⟩`}` `{}` 0.19

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

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

optional `procnamestyle=`⟨*style*⟩ `keywordstyle` 0.19

defines the style in which procedure and function names appear.

optional `indexprocnames=`⟨`true`|`false`⟩ `false` 0.19

If activated, procedure and function names are also indexed.

To do: The `procnames` aspect is still unsatisfactory (since unchanged for more than a year). It marks (and indexes) only the function definitions so far, but it would be possible to mark also the following function calls. A key `incremetalprocnames=`⟨`true`|`false`⟩ could control whether function names are added to a special keyword class, which 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. Globally?

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.

new, optional `hyperref={`⟨*identifiers*⟩`}` 0.21

new, optional `morehyperref={`⟨*identifiers*⟩`}` 0.21

new, optional `deletehyperref={`⟨*identifiers*⟩`}` 0.21

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

5.4 Literate programming

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

```
var i:integer;

if (i<=0) i := 1;
if (i>=0) i := 0;
if (i<>0) i := 0;
```

```
\begin{lstlisting}{}
var i:integer;

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

Funny, isn't it? We could write `i := 0` respectively i ← 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*⟩ 0.20

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
```

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.

new, optional `lgrindef=`⟨*language*⟩ 0.21

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.

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 (probably 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 be 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. The format definition

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

activated via `\lstset{format=C}` yields

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

```
\begin{lstlisting}{}
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}
```

with the following result

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

```
\begin{lstlisting}{}
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}`.

6 Forthcoming

I'd like to support more languages, for example Maple, PostScript, Reduce, and so on. Fortunately my lifetime is limited, so other people may do that work. Please (e-)mail me your language definitions.

Some people made suggestions to extend the functionality. There things aren't listed here. Feel free to email me your suggestions.