Listings.dtx Version 0.2000☺

Abstract

Listings.dtx is a source code printer for LaTeX. You can typeset stand alone files as well as enter listings using an environment similar to `verbatim` as well as you can print fragments using a command similar to `\verb`. Many parameters control the output and the package supports a wide spectrum of programming languages — some come already with `lstdrvrs.dtx`.

Minor incompatibilities! And again there will be teething troubles since some sensitive parts have been rewritten.

Preface

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

Alternatives. This package is certainly not the final utility for typesetting source code. Other programs do their jobs very well if you are not satisfied with listings. Mainly I should mention the 'general text to PostScript' converter a2ps and the 'source code to `.tex`' cross compiler LGrind. Both's functionality can be compared with the functionality of this package. Therefore they are too complex to describe them here more detailed.

The listing package — note the missing `s` — is not worth to talk about in our context since it defines `\listoflistings` and an environment without any keyword, comment, string or whatever else detection.[1] But it's possibly useful if you have another tool doing that work.

The algorithms package (`algorithmic.sty` and `algorithm.sty`) goes a quite different way. You describe an algorithm and the package *formats* it! 'Ruled', 'boxed' and floating algorithms, a list of algorithms and line numbers are also supported.

To complete the alternatives, here some packages for verbatim listings, which do not pretty-print the source code — at least not automatically. verbatim and moreverb together provide verbatim listings, verbatim listings of stand alone files, verbatim output to a file, 'boxed' verbatims and line numbers. The alltt package is like verbatim except that `\`, `{` and `}` have the usual meanings: You can use commands in the verbatims, e.g. select different fonts or enter math mode. Finally I'd like to mention fancyvrb. Roughly speaking it's a super set of the other three packages, but many more parameters control the output. Moreover there exists an interface to listings, i.e. you can pretty-print source code automatically.

All packages are available from CTAN.

Reading this manual. If you are experienced with the listings package, you should read the paragraph "*News and changes*" below. If you've just decided to try or to use this package (possibly together with an alternative listed above), read section "*1. Getting started*" step by step. Afterwards and after some practice with the most basic features you should be able to pick up more from the main reference.

News and changes. We begin with some new features. The 'escape', 'frame' and 'fancyvrb' aspects have been extended and automatic breaking of long lines is completely new. Second order keywords need no extra loading since they are built-in. About 30 keys have been added, look in sections 2 and 3 for the label '0.20' in the right margin. In particular listings might have captions and can float. Note also that `\lstinline` has an optional key=value list and that `\lstbox` doesn't exist any more. The package uses an auto-detection. Eventually I hope that the package works under Lambda.

[1] Furthermore the aphorism about documentation at the end of `listing.sty` is not true.

Now come commands and keys which have been renamed or replaced (often by more genereal ones).

`\lstlistoflistings`	was	`\listoflistings`
`\lstlistlistingname`	was	`\listlistingsname`
`identifierstyle`	was	`nonkeywordstyle`
`directives`	was	`cdirectives`
`keywordcommentsemicolon`	was	`doublekeywordcommentsemicolon`
`advancelabel`	was	`advancelineno`
`basewidth`	replaces	`baseem`
`stringspaces`	replaces	`blankstring`
`\lst@definelanguage`	replaces	`\lstdefinedrvlanguage`

By definition `\lststorekeywords`, `pre`, `post`, `\lstname` and `\lstintname` are obsolete. Read section 2.15 if you want to know more.

Mailing list. If you want to receive release information, bug reports, workarounds, and so on, email to `cheinz@gmx.de` with subject `subscribe listings`. In the body you might state which information you don't want.

Thanks. There are many people I have to thank for fruitful communication, posting their ideas, giving error reports (first bug finder is listed), adding programming languages to `lstdrvrs.dtx`, and so on. If you want to know that in detail, search for the names in the implementation part. 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, Peter Bartke, Peter Biechele, Kai Below, David Carlisle, Patrick Cousot, Holger Danielsson, Detlev Dröge, Anders Edenbrandt, David John Evans, Harald Harders, Christian Haul, Aidan Philip Heerdegen, Jürgen Heim, Dr. Jobst Hoffmann, Torben Hoffmann, Berthold Höllmann, Marcin Kasperski, Knut Müller, Torsten Neuer, Heiko Oberdiek, Zvezdan V. Petkovic, Michael Piotrowski, Manfred Piringer, Ralf Quast, Aslak Raanes, Detlef Reimers, Magne Rudshaug, Andreas Stephan, Gregory Van Vooren, Dominique de Waleffe, Michael Weber, Herbert Weinhandl, Michael Wiese, Jörn Wilms and Kai Wollenweber.

I hope this list is complete.

User's guide

1 Getting started

1.1 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 'language driver' or short 'driver' is generated from `lstdrvrs.dtx`.

Copyright. The listings package is copyright 1996–1999 Carsten Heinz. The language drivers are copyright 1997/1998/1999 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, at your option, 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 document and sell the 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.

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

1.2 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`	→	`texmf/source/latex/listings`
`lstdrvrs.dtx`	→	`texmf/source/latex/listings`

   Of course, you need not to use the TDS. Simply adjust the directories below.

2. Remove all files from `texmf/tex/latex/listings` or create that directory if you don't update from an earlier version.

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   →  texmf/tex/latex/listings
   lstlang1.sty, lstlang2.sty  →  texmf/tex/latex/listings
   listings.cfg                →  texmf/tex/latex/listings
   ```

5. If your TEX implementation uses a filename database, update it.

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

You might need to increase save stack size or string memory since listings 0.20 needs lots of them. But that depends a bit on the used features.

1.3 A minimal file

Please read section 2.1 before you use this package in a real document. Here is some kind of minimal file.

```
\documentclass{article}

\usepackage{listings}
\lstloadlanguages{C++,Pascal}% if both are needed

\begin{document}

\lstset{language=Pascal}% select Pascal
% Any example can be inserted here.

\end{document}
```

We load the listings package and then C++ and Standard Pascal language drivers. This selects neither Pascal nor C++. Later in the document Pascal becomes active.

1.4 Typesetting listings I

You can print stand alone files or source code you typed directly into a `.tex` file, where you have the choice between 'displayed' listings or code fragments within a paragraph. In imitation of LATEX's `\verb` command you can write `\lstinline!var i:integer;!` and get '**var** i:**integer**;'. The exclamation marks delimit the code fragment and could be replaced by any character not in the code fragment, i.e. `\lstinline$var i:integer;$` would produce the same output.

The `lstlisting` environment typesets the source code in between. It has one optional and one name parameter, which we leave empty for the moment.

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

Write('Keywords are case ');
WritE('insensitive here.');
```

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

Write('Keywords are case ');
WritE('insensitive here.');
\end{lstlisting}
```

You might take the LaTeX source code you see on the right, put it into the minimal file, and run it through a TeX compiler. You'll get an output similar to the result shown on the left. The optional argument 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('Keywords are case ');
WritE('insensitive here.');
\end{lstlisting}
```

Note that the specified lines must exist, or you will get a "runaway argument" error. The frame shows that empty lines at the end of a listing are not printed. If you definitely what the line 5 here, read about `showlines` in section 2.3.

The command `\lstinputlisting` also has one optional and one file name argument. It pretty-prints the stand alone file:

```
\lstinputlisting{listings.tmp}
```

```
\lstinputlisting{listings.tmp}
```

Do you wonder about the left-hand side? Well, the file `listings.tmp` contains the current/last example. This is exactly the line you see on the right. If you want this line to be typeset in Pascal mode, you get what you've got.

1.5 The "key=value" interface

The listings package uses `keyval.sty` from the graphics bundle by David Carlisle. Each 'parameter' is controlled by an associated key and a user supplied value. The command `\lstset` gets a "key=value" as argument. You have seen this in the minimal file and in the last section. You can set more than one parameter with a single `\lstset` if you separate two or more key=value pairs by commas like this:

```
\lstset{language=Pascal,keywordstyle=\bfseries}
```

If the value itself contains a comma, you must enclose the value in braces, for example `\lstset{keywords={one,two,three}}`. Without the additional braces the keyval package would set the one and only keyword `one`, and then give an error message since the keys `two` and `three` do not exist.

`\lstinputlisting`, `\lstinline` and the `lstlisting` environment all have an optional argument. If you use a key=value list as optional argument, these selections are valid for the particular listing only and the previous values are restored afterwards. For example, if the current language is Pascal, but you want `testfile.f95` from line 3 on, write

```
\lstinputlisting[first=3,language=Fortran]{testfile.f95}
```

Afterwards Pascal is still active. This principle applies to all keys.

Some keys also have optional arguments. If you want to use such an optional argument inside the optional argument of a pretty-printing command or environment, you must put braces around the whole value. You'll find an example on page 8.

Finally you should know that there are different kinds of keys. Some like `first` and `last` make sense only if they are used for a single listing, and therefore used inside an optional argument. Other keys don't worry about whether they are set via `\lstset` or via an optional arguments. Furthermore some keys have default values, e.g. `flexiblecolumns` without any `=true` turns flexible columns on. But you must use `=false` if you want to turn them off.

1.6 Typesetting listings II

You already know all pretty-printing commands and environments. You need to learn the parameters which control how the listings are printed. First you'll have to ensure that your source code can be processed at all. Please read section 2.11 if you use national characters inside listings, i.e. characters with (ASCII) codes greater than 127. Otherwise you'll get really funny results.

You might get some unexpected output if your source code contains a tabulator. The package assumes tabulator stops at columns 9, 17, 25, 33, and so on. This is pre-defined via `\lstset{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}
```

Note that the left-hand side is printed with `tabsize=2`, but the verbatim code uses `tabsize=4`. Tabulators can be made visible via the `visibletabs` key, see section 2.6.

Another special character is a form feed causing an empty line. If you want a new page every form feed, write `\lstset{formfeed=\newpage}`.

Listing: A floating example

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

Write('Keywords are case ');
WritE('insensitive here.');
```

Another item is about how source code looks like in your `.tex` file. You might want to use indention which must be removed to pretty-print a listing. 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('Keywords are case ');
WritE('insensitive here.');
```

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

␣␣␣Write('Keywords␣are␣case␣');
␣␣␣WritE('insensitive␣here.');
\end{lstlisting}
```

Note that empty lines as well as the beginning and the end of the environment need not to respect the indention. Moreover note that tabulators expand to `tabsize` spaces before we gobble. If we don't gobble characters any more, tabulators don't expand to spaces.

Something different at the end of this section. By default listings do not float, but look at this:

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

Write('Keywords are case ');
WritE('insensitive here.');
\end{lstlisting}
```

The empty optional argument for `caption` prevents the package from using a listing number. The key `float` is not boolean. It is used without any value or gets a subset of `tbph` which determines the placement of the float. This is similar to the standard `figure` and `table` environments. If you put the example into the minimal file and run it through TEX, please don't wonder: You'll miss the horizontal rules since they are described in section 2.9.

1.7 Figure out the appearance

Keywords are typeset bold, comments in italic shape and spaces in strings appear as ␣. You can change that default behaviour, for example

```
\lstset{
    basicstyle=\small,          % print whole listing small
    keywordstyle=\color{red}\bfseries
                \underbar,      % underlined bold red keywords
    identifierstyle={},         % nothing happens to other identifiers
    commentstyle=\color{white}, % white comments
    stringstyle=\ttfamily,      % typewriter type for strings
    stringspaces=false}         % no special string spaces
```

We typeset a previous example with these selections again.

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

end;

Write('Keywords are case ');
WritE('insensitive here.');
```

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

Write('Keywords are case ');
WritE('insensitive here.');
\end{lstlisting}
```

The style definition above uses two different kinds of commands. On the one hand `\ttfamily` and `\bfseries` both take no arguments, and on the other `\underline` gets exactly one argument. In general the *very last* token of `keywordstyle` and `identifierstyle` *might* be a macro getting exactly one argument, namely the (non)keyword. All other tokens *must not* take any arguments — or you will get deep in trouble.

Warning: You shouldn't use striking styles too often, but 'too often' depends on how many keywords the source code contains, for example. Your eyes would concentrate on the framed, bold red printed keywords only, and your brain must compensate this. Reading such source code could be very exhausting. If it were longer, the last example would be quite good in this sense. Believe me.

1.8 Line numbers

There are three stages: Print line numbers, control the printed line numbers, and refer to line numbers. Let us try to print tiny numbers, each second line, 5pt distance to the listing:

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

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

Note that `labelstep=0` turns line numbering off and `labelsep=10pt` is default. However, in the sequel we use the selections from above even if it doesn't appear in the verbatim part.

The `lstlisting` environment allows you to interrupt your listings. Remember that the 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('Keywords are case ');
WritE('insensitive here.');
\end{lstlisting}
```

The next `Test` listing goes on with line number 8. Note that the empty line at the end of the first part is not printed, but it counts for line numbering. This continue mechanism has two exceptions: An empty (= `{}`) named listing always starts with line number one, no matter whether line numbers are printed or not. 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:

```
\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('Keywords are case ');
WritE('insensitive here.');
\end{lstlisting}
```

Finally you should know how to reference line numbers. You need one character not used otherwise in the source code. In the example we use the percent character which must be entered as `\%`. After `\lstset{escapechar=\%}` you can put TEX material between two percent characters, for example `\label{comment}`.

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

```
\lstset{escapechar=\%}%     <===

\begin{lstlisting}{}
for i:=maxint to 0 do
begin
    { do nothing }%\label{comment}%
end;
\end{lstlisting}
Line \ref{comment} \ldots
```

You should put `%\label{`whatever`}%` into a comment if you want to reference line numbers of a stand alone file. Or your compiler/interpreter will have problems.

1.9 Indent the listing

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

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

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

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

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

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

Note that `\lstset{indent=15pt}` also changes the indent for the second listing. If you want to indent a single listing, use the optional argument of the environment or input command.

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. Refer section 2.8 for a description of `spread`.

1.10 Fixed and flexible columns

The first thing a reader notices is — except different styles for keywords, etc. — the column alignment of a listing. The problem: I don't like listings in typewriter type and other fonts need not to have a fixed width. But we don't want

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

only because spaces are not wide enough. There is a simple trick to avoid such things. We make boxes of the same width and put one character in each box:

i	f		x	=	y		t	h	e	n		w	r	i	t	e		...
							e	l	s	e		p	r	i	n	t		...

Going this way the alignment of columns can't be disturbed. But if the boxes are not wide enough, we get 'i f x=y then ...', and choosing the width so that the widest character fits in leads to 'i f x = y t h e n w r i t e ...'. Both are not acceptable. Since all input will be cut up in units, we can 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	

. 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 text 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. To go around this problem the listings package supports more 'flexible columns' in contrast to the fixed columns above. 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`
`  MEN`
`WOMEN are`
`better MEN`
	WOMEN are
	MEN
	WOMEN are
	better MEN
		WOMEN are
		MEN
		WOMEN are
		better MEN

Hope this helps. Note the varying numbers of spaces between 'WOMEN' and 'are' and look at the different outputs. The flexible column format typesets all characters at their natural width. In particular characters never overlap. If a word needs more space than reserved ('WOMEN'), the rest of the line moves to the right. Sometimes a following word needs less space than reserved, or there are spaces following each other. Such 'surplus' space is used to fix the column alignment: The two blanks in the first line came out as a single space. You can see this since the single space in the third line has been printed properly. We can show all this more drastic if we reduce the width of a single character box:

```
                        0.3em          0.0em
WOMEN  are      WOMEN are      WOMEN are
       MEN          MEN            MEN
WOMEN are       WOMEN are      WOMEN are
better MEN      better MEN     better MEN
```

In flexible column mode the first 'MEN' moves to the left since the blanks before are $7 \cdot 0.0\text{em} = 0\text{em}$ wide. Even in flexible mode you shouldn't reduce 'base width' to less than 0.33333em ($\approx$ width of a single space in some fonts).

You want to know how to select the flexible column format and how to reset to fixed columns? Try `flexiblecolumns` and `flexiblecolumns=false`. The predefinition of the 'base width' is `basewidth={0.6em,0.45em}`, where the first width is for fixed mode and the second for flexible columns. Change it if you like.

1.11 Selecting other languages

You already know that `language=`⟨*language name*⟩ selects programming languages — at least Pascal. But that's not the whole truth. Some languages know different dialects (= version or implementation), for example Fortran 77 and Fortran 90. You can choose such special versions with the optional argument of `language`. Write

```
\lstset{language=[77]Fortran}% Fortran 77
\lstset{language=[XSC]Pascal}% Pascal XSC
```

to select Fortran 77 and Pascal XSC, respectively. Remember that you must put braces around the value if you select a language with the optional arguments.

Table 1 shows all languages and dialects supported by `lstdrvrs.dtx`. Use the given names as (optional) values to `language`. An 'empty' language is also defined: `\lstset{language={}}` detects no keywords, no comments, no strings, and so on. Each first dialect is default dialect, e.g. `\lstset{language=C}` selects ANSI C. After `\lstset{defaultdialect=[Objective]C}` Objective-C is default dialect for C, but the language is not selected with the command. Note that predefined default dialects change from release to release. Thus: Always define (after package loading) the dialects you use as default dialects.

Remark: The languages have all bugs coming from the language defining commands described in section 2.14, e.g. in Ada and Matlab it is still possible that the package assumes a string where none exists.

Table 1: Pre-defined languages

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

2 Main reference

In this section we list all user environments, commands and keys together with their parameters and possible values. Parts dealing with yet unknown features should always contain examples. If not write your own ones. The numbers in the right margin give the version number of introduction (either as internal or as user macro/key). The labels on the left give you some important information about the key or command. For example, *addon* indicates additional functionality. The labels should be clear.

`\lstset{`⟨*key=value list*⟩`}` 0.19

sets the values of the specified keys, see section 1.5.

2.1 Package loading

As usual in LaTeX the package is loaded by `\usepackage[`⟨*options*⟩`]{listings}`, where `[`⟨*options*⟩`]` is optional. Each option loads a 'listings-aspect' (= collection of commands and keys) or prevents the package from loading it if the aspect name is preceded by an exclamation mark. But even in the latter case an aspect is

loaded later if a pre-defined programming language needs it. See section 6.6 for a complete list of aspects. Here are the main ones.

`0.17` Use this option to compile documents created with version 0.17 of the listings package. Note that you can't use old driver files and that the option *does not guarantee full compatibility.*

`0.19` to compile documents created with version 0.19.

`rdkeywords`, `breaklines`, `index` and `procnames` define the keys of the aspect(s). It will be obvious which *optional* marked keys belong to which aspect.

`fancyvrb` This option is different since the key `fancyvrb` is always defined and loads some definitions on demand. The option loads these definitions at package loading which is faster than loading it later. If you use this option, the fancyvrb package must already be loaded!

It makes no sense to write `\usepackage[!0.17]{listings}` to 'unload' the compatibility mode since it isn't loaded by default anyway. But if you don't use line numbering, you can write `\usepackage[!labels]{listings}` to save some TEX memory. However, you can load it later with `\lstloadaspects{labels}`.

new `\lstloadaspects{`⟨*comma separated list of aspect names*⟩`}` 0.20

loads the specified aspects if they are not already loaded.

After package loading I strongly recommend to load all used dialects of programming languages with the following command, which can be used in the preamble only (before `\begin{document}`).

preamble `\lstloadlanguages{`⟨*comma separated list of languages*⟩`}` 0.19

It loads all specified languages, where each language is given in the form `[`⟨*dialect*⟩`]`⟨*language*⟩. Write `[Visual]C++` if you use Visual C++; `C++` would load ANSI C++ only.

new,data `\lstdriverfiles` 0.20

contains a comma separated list of the language driver file names. This macro is usually defined in a configuration file.

Finally note that the float package must be loaded before listings.

2.2 Languages and styles

We distinguish programming languages and styles used to print a listing. Since both are defined in terms of key=value lists, it is possible to use a style key inside a language definition or vice versa. However, the pre-defined languages don't use style keys.

`language={`⟨*dialect*⟩`}`⟨*language name*⟩ 0.17

activates a (dialect of a) programming language. The arguments are case insensitive and spaces have no effect.

`defaultdialect={`⟨*dialect*⟩`}`⟨*language*⟩ 0.19

defines a default dialect for a language, that means a dialect which is selected whenever you leave out the optional argument. 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`.

Note that a configuration file possibly defines some default dialects.

`style=`⟨*style name*⟩ 0.18

activates a style. The argument is case insensitive.

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

stores the key=value list. You can select the style via `style=`⟨*style name*⟩.

To do: It's easy to crash the package with `style` — and also with `language`. Write `\lstdefinestyle{crash}{style=crash}` and `\lstset{style=crash}`. TEX's capacity will exceed, sorry [parameter stack size]. Only bad girls use such recursive calls, but only good girls use this package. Thus the problem is of minor interest.

2.3 Typesetting listings

First come the pretty-printing commands and environments. They all have an optional ⟨*key=value list*⟩ which modify parameters for the specific listing only. Note that empty lines at the end of listings are always dropped, but they count for line numbering.

`\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.

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

typesets the code between `\begin{lstlisting}` (+ arguments + line break) and `\end{lstlisting}`. Source code right before and LATEX code after the end of environment is typeset (if nonempty line) respectively executed.

Same named listings have common line counter, 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 (= `{ }`).

addon `first=`⟨*line number*⟩ 0.1

addon `last=`⟨*line number*⟩ 0.1

determine the printed line range of stand alone files and environments. They must be used on individual listings in the optional arguments. Default range is 1–9999999.

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

controls whether the command respectively environment typeset the listings. 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 in spite of that.

new `showlines`⟨*true*|*false*⟩ or `showlines` 0.20

If true, the package prints empty lines at the end of listings — no matter what I've said before.

new `gobble=`⟨*number*⟩ 0.19

gobbles ⟨*number*⟩ characters at the beginning of each line. If necessary, tabulators expand to `tabsize` spaces before they are gobbled. Code lines with less than ⟨*number*⟩ characters are considered to be empty.

If you use this key together with the environment, `\end{lstlisting}` need not to be indented by ⟨*number*⟩ spaces. The package will find the end.

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

makes sense with individual 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 appears. If you use the key without a value, it uses the placement specifier `tbp`.

2.4 Captions

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

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

can only be used on individual listings. 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.

You may use `\label` inside ⟨*caption text*⟩ and elsewhere `\ref` to refer to the listing. This makes sense only if ⟨*short*⟩ is not empty.

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

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

specifies the position(s) of the caption. After loading the package it puts captions at the top of listings.

renamed `\lstlistoflistings` 0.16

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

renamed `\lstlistlistingname` 0.16

contains `Listings`, the header name for the list of listings.

new,data `\lstlistingname` 0.20

contains `Listing`. It's the string used to label the caption, see page 8.

new,data `\thelstlisting` 0.20

prints the caption's label number. The default definition depends on whether the document class supports chapters. It's either `\arabic{lstlisting}` or `\thechapter.\arabic{lstlisting}`, but nonpositive chapter numbers are not printed.

new `abovecaptionskip=`⟨*skip*⟩ 0.20

new `belowcaptionskip=`⟨*skip*⟩ 0.20

is the vertical skip above respectively below each caption. The pre-definition is `\smallskipamount`.

2.5 Labels

`labelstep=`⟨*step*⟩ 0.16

No labels are printed if ⟨*step*⟩ is zero, which is the pre-definition. Otherwise all lines such that "line number $\equiv$ 0 modulo ⟨*step*⟩" get a label, which appearance is controlled by `labelstyle` and `\thelstlabel`.

`labelstyle=`⟨*style*⟩ 0.16

determines the font and size of the labels. It is pre-defined to be empty.

new,data `\thelstlabel` 0.20

prints the label numbers of the lines. `\arabic{lstlisting}` is the default definition.

`labelsep=`⟨*dimension*⟩ 0.19

is the distance between label and listing. 10pt is default separation.

new `firstlabel=`⟨*number*⟩ 0.20

renamed `advancelabel=`⟨*number*⟩ 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`. However, if you put the verbatim part into the minimal file, you won't get the result shown on the left.

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

```
\renewcommand*\thelstlabel
    {\oldstylenums{%
          \the\value{lstlabel}}}
\begin{lstlisting}[firstlabel=753]{}
begin { 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!

2.6 Figure out the appearance

`basicstyle=`⟨*basic style and size*⟩ 0.18

`keywordstyle=`⟨*style for keywords*⟩ 0.11

`ndkeywordstyle=`⟨*style for second order keywords*⟩ 0.19

optional `rdkeywordstyle=`⟨*style for third order keywords*⟩ 0.19

renamed `identifierstyle=`⟨*style*⟩ 0.18

`commentstyle=`⟨*style*⟩ 0.11

`stringstyle=`⟨*style*⟩ 0.12

new,optional `texcsstyle=`⟨*style*⟩ 0.20

new,optional `directivestyle=`⟨*style*⟩ 0.20

Each value of these keys determines the font and size (or more general style) in which special parts of a listing appear. The *last* token of (nd,rd) keyword and identifier style might be an one-parameter command like `\textbf` or `\underline`.

The package uses keyword style if TEX control sequences or compiler directives are defined but no style is specified.

`stringspaces=`⟨*true*|*false*⟩ 0.12

lets blank spaces in strings appear ␣ or as blank spaces. The first (=true) is pre-defined.

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

lets all blank spaces appear ␣ or as blank spaces. The latter case (=false) is pre-defined.

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

make tabulators visible or invisible (default). A visible tabulator looks like ␣———, but that can be changed. If you choose invisible tabulators but visible spaces, tabulators are converted to appropriate number of spaces.

new `tab=`⟨*token sequence*⟩ 0.20

⟨*token sequence*⟩ 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*⟩ 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. It is initialized with `tabsize=8`.

`formfeed=`⟨*token sequence*⟩ 0.19

Whenever a listing contains a form feed ⟨*token sequence*⟩ is executed. It is initialized with `formfeed=\bigbreak`.

2.7 Language specific style keys

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

prints or drops PODs in Perl.

new,optional `usekeywordsinside=`⟨*true*|*false*⟩ 0.20

The package either use the first order keyword list for HTML or prints all identifiers inside `<>` in keyword style. The first case is selected by default.

new,optional `makemacrouse=`⟨*true*|*false*⟩ 0.20

is true by default: Macro use of identifiers defined as first order keywords also prints the surrounding `$(` and `)` in keyword style. E.g. you could get **$(strip** $(BIBS)**)**. If deactivated you would get $(**strip** $(BIBS)).

2.8 All about listing alignment

We start with the alignment of listings and surrounding text.

`indent=`⟨*dimension*⟩ 0.19

indents each listing by ⟨*dimension*⟩, which is initialized with 0pt. This command is the best way to move line numbers (and the listing) to the right.

`wholeline=`⟨*true*|*false*⟩ 0.19

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

bug `spread=`⟨*dimension*⟩ or `spread={`⟨*inner*⟩`,`⟨*outer*⟩`}` 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. It is initialized via `spread=0pt`. 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`.

`lineskip=`⟨*additional space between lines*⟩ 0.17

specifies the additional space between lines in listings. You may write `lineskip=-1pt plus 1pt minus 0.5pt` for example, but 0pt is the default.

`boxpos=`⟨`b`|`c`|`t`⟩ 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, top baseline or centered.

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

Now we go on with column alignment inside listings.

`outputpos=`⟨`c`|`l`|`r`⟩ 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 resp. | listing |, |listing | and | listing| with flexible columns (using pre-defined base widths). By default the output is centered.

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

selects the flexible respectively the fixed column format, refer section 1.10.

new `basewidth=`⟨*width*⟩ or `basewidth={`⟨*fixed*⟩`,`⟨*flexible mode*⟩`}` 0.16

sets the width of a single character box for fixed and flexible column mode (both to the same value or individually). `basewidth={`0.6em`,`0.45em`}` is the pre-definition.

new `fontadjust=`⟨*true*|*false*⟩ or `fontadjust` 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 2.6 make heavy font size changes, see the example below.

If you prefer the LGrind package rather than listings (I can't imagine that ;–), you should try `basewidth=1ex` together with `flexiblecolumns` and `fontadjust`, but you have to play a bit with the base width.

```
{ 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!

2.9 Frames

`frame=`⟨*any 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, but as optional argument or argument to `\lstset`, of course. 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 number inside frames.

`framerulewidth=`⟨*dimension*⟩ 0.19

`framerulesep=`⟨*dimension*⟩ 0.19

These keys control the width of the rules and the space between double rules. Pre-defined values are .4pt width and 2pt separation.

`frametextsep=`⟨*dimension*⟩ 0.19

controls the space between frame and listing. The pre-defined value is 3pt.

new `framespread=`⟨*dimension*⟩ 0.20

makes the frame on each side half ⟨*dimension*⟩ wider. It is initilaized with 0pt.

new `frameround=`⟨`t`|`f`⟩⟨`t`|`f`⟩⟨`t`|`f`⟩⟨`t`|`f`⟩ 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 set via `\thinlines` and `\thicklines`.

Note: The size of the quarter circles is independent from `framespread`. The size is possibly adjusted to fit LaTeX's circle sizes.

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

```
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}
```

new `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.

⟨*left shape*⟩ and ⟨*right shape*⟩ are both '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` for both shapes.

⟨*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 + corners, 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` most outer.

Above I 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.

2.10 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` 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*⟩ 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 a nice way.

`escapechar=`⟨*single 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=\%`.

new `escapeinside=`⟨*single character*⟩⟨*single 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.

new `escapebegin=`⟨*'begin' tokens*⟩ 0.20

new `escapeend=`⟨*'end' tokens*⟩ 0.20

The tokens are executed at the beginning respectively at the end of each escape, in particular for `texcl`. See section 2.11 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.

2.11 National characters

You probably want to use national characters in your listings, for example in comments. The first possibilty makes use of `escapechar` or `escapeinside`:

ä è ī œ ŭ

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

A better way is to type in the national characters directly:

`extendedchars=`⟨*true*|*false*⟩ or `extendedchars` 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 — read section 4.1 if you use the latter one.

However, the extended character tables don't cover Arabic, Chinese, Hebrew, Japanese, and so on. In this case you'll have to use Λ (Lambda), the LaTeX pendant to Omega. The keys `escapebegin` and `escapeend` allows you to select and deselect Ω compiled translation processes. Then the most comfortable way of usage are comment lines.

```
\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}
```

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}
```

Finally note that the 'interface' to Λ is completely untested.

2.12 Environments

The command used to define the `lstlisting` environment is public now. The syntax comes from LaTeX's `\newenvironment`.

`\lstnewenvironment{`⟨*name*⟩`}[`⟨*number of parameters*⟩`][`⟨*opt. default arg.*⟩`]` 0.19
`{`⟨*starting code*⟩`}{`⟨*ending code*⟩`}`

We present two examples, namely `lstlisting` and version 0.17 `listing` environment. 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 and an error message is issued. 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}
```

The package defines one more (optional) environment for documentation. It's the environment which typesets verbatim code on the right and the result on the left (or the result atop the code if the verbatim code is too wide). The definition is a bit longer, needs to be explained, and is therefore not shown here.

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

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

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

2.13 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 also possible, but you must find somebody who implements them ; –). The fancyvrb package is available from `CTAN: macros/latex/contrib/supported/fancyvrb`.

`fancyvrb=`⟨*true*|*false*⟩ 0.19

activates or deactivates the interface. This defines an appropriate version of `\FancyVerbFormatLine` to make the two packages work together. If active, the verbatim code read by the fancyvrb package is typeset by the listings package, i.e. with emphasized keywords, strings, comments, and so on. — You should know that `\FancyVerbFormatLine` is responsible for typesetting a single code line.

If fancyvrb and listings provide similar functionality, use fancyvrb's.

This second interface works with `Verbatim`, `BVerbatim` and `LVerbatim`. But you shouldn't use `defineactive`. (As far as I can see it doesn't matter since it does nothing at all.)

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

```
\lstset{commentline=\ }% :-)
\fvset{commandchars=\\\{\}}

\begin{BVerbatim}
First verbatim line.
\fbox{Second} verbatim line.
\end{BVerbatim}
\vskip72.27pt

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

The last two lines are wider than the first two since the default `basewidth` equals not the width of a single typewriter type character.

2.14 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={--}{!}`.

addon `\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 which might changes.

The ⟨*key=value list*⟩ is executed (additionally) when you select the language. The last optional argument should specify which `lst`-aspects (see section 6.6) the language definition requires. For example, ANSI C uses `keywords`, `comments`, `strings` and `directives`.

`\lst@definelanguage` (same syntax) defines languages in driver files.

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

defines an alias for a programming language. Any dialect of ⟨*alias*⟩ selects in fact the same dialect of ⟨*language*⟩. It's also possible to define an alias for one dialect: `\lstalias[`⟨*dialect alias*⟩`]{`⟨*alias*⟩`}[`⟨*dialect*⟩`]{`⟨*language*⟩`}`. Here all four parameters are *non*optional. 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 configuration file possibly defines some aliases.

Now come all the language keys, which might be used in the key=value list of `\lstdefinelanguage`. Note: *If you want to enter* `\`, `{`, `}`, `%`, `#` *or* `&` *inside or as an argument here, you must do it with a preceding backslash!*

`keywords={`⟨*keywords*⟩`}` 0.11

`morekeywords={`⟨*additional keywords*⟩`}` 0.11

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

`ndkeywords={`⟨*second order keywords*⟩`}` 0.19

`morendkeywords={`⟨*additional second order keywords*⟩`}` 0.19

`deletendkeywords={`⟨*second order keywords to remove*⟩`}` 0.19

optional `rdkeywords={`⟨*third order keywords*⟩`}` 0.19

optional `morerdkeywords={`⟨*additional third order keywords*⟩`}` 0.19

optional `deleterdkeywords={`⟨*third order keywords to remove*⟩`}` 0.19

Each ⟨*keywords*⟩ like value (here and below) is a list of keywords separated by commas. `keywords={save,Test,test}` defines three keywords (if keywords are case sensitive). Use `keywords={}` to remove all first order keywords.

`!"#$%&'()*+-./0123456789:;<>?@[]^_‘{|}~` can all be used in keywords, but note that you must write `\#`, `\%`, `\&`, `\\`, `\{` and `\}` instead of `#`, `%`, `&`, `\`, `{` and `}`. Please read the notes about the ‘`also`’ keys if you use unusual charaters in keywords.

Note that there is the key `texcs` to define control sequences as keywords.

`sensitive=`⟨*true*|*false*⟩ 0.14

makes the keywords (first, second and third) case sensitive resp. insensitive. This key affect the keywords only in the phase of typesetting. In all other situations keywords are case sensitive, i.e. `deletekeywords={save,Test}` removes ‘save’ and ‘Test’, but neither ‘SavE’ nor ‘test’.

`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. Note that the auto-detection might fail if you remove keywords.

new `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.

`stringtest=`⟨*true*|*false*⟩ 0.19

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

`stringizer=[`⟨*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.

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

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

defines/expands the list of control sequences for TEX and LATEX.

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

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

new,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.

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

addon `commentline=`⟨*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.

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

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).

addon `singlecomment={`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}` 0.13

addon `doublecomment={`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}` 0.13

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 need 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.

addon `nestedcomment={`⟨*delimiter*⟩`}{`⟨*delimiter*⟩`}` 0.13

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

renamed,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.

2.15 Obsolete keys and commands

We come to the obsolete features — obsolete by definition. Don't use these keys and commands any more. In the worst case other keys must be introduced. This might sound strange but in particular the arguments of `pre` and `post` are hardly to control. Therefor they are deactivated in some cases, and this means that all features defined via these keys are deactivated. That's not good.

obsolete `pre=[`⟨*continue*⟩`]{`⟨*commands to execute*⟩`}` 0.12

obsolete `post=[`⟨*continue*⟩`]{`⟨*commands to execute*⟩`}` 0.12

The given control sequences are executed before and after typesetting resp. when continuing a listing, but in all cases inside a group. The commands are not executed for `\lstinline` or if the package makes an extra `\hbox` around the listing. The reason is that the user given pre and post commands are assumed to be unsave inside `\hbox`. By default ⟨*continue*⟩ equals ⟨*commands to execute*⟩. All arguments are pre-set empty.

obsolete `\lststorekeywords`⟨*macro*⟩`{`⟨*keywords*⟩`}` 0.18

stores ⟨*keywords*⟩ in ⟨*macro*⟩ for use with keyword keys. This command can't be used in a language definition since it is a command and not a key.

obsolete `\lstname` 0.19

contains the name of the current (last) listing in *printable* form.

obsolete `\lstintname` 0.19

contains the name of the current (last) listing possibly in nonprintable form.

3 Experimental features

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

3.1 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`. Finally 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[2] if the current language is Pascal. Now environment examples:

```
 !"#$%&'()*+,-./
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}}
```

3.2 Export of identifiers

It would be nice to export function or procedure names, for example to index them automatically or to use them in `\lstlistoflistings` instead of a listing name. In general that's a dream so far. The problem is that programming languages use

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

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.

optional `index={`⟨*identifiers*⟩`}` 0.19

new,optional `ndindex={`⟨*identifiers*⟩`}` 0.20

⟨*identifiers*⟩ is a comma-separated list of identifiers. Each appearance of such an identifier is indexed.

optional `indexmacro=`⟨*'one parameter' macro*⟩ 0.19

new,optional `ndindexmacro=`⟨*'one parameter' macro*⟩ 0.20

The specified macro gets exactly one parameter, namely the (nd)identifier, and must do the indexing. (`nd`)`indexmacro=\lstindexmacro` is the predefinition and we have

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

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

⟨*keywords*⟩ is a comma-separated list of keywords, which indicate 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 for procedure names*⟩ 0.19

defines the style in which procedure and function names appear.

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

If activated, procedure and function names are also indexed (if used with `index` option).

3.3 Automatic line breaking

new,optional `breaklines=`⟨*true*|*false*⟩ or `breaklines` 0.20

activates or deactivates automatic line breaking of long lines. This is deactivated by default.

new,optional `breakindent=`⟨*dimension*⟩ 0.20

indents the second, third, ... line of broken lines by ⟨*dimension*⟩. It is initialized with 20pt.

new,optional `breakautoindent=`⟨*true*|*false*⟩ or `breakautoindent` 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. It is activated by default.

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

new,optional `prebreak=`⟨*tokens*⟩ 0.20

new,optional `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. Both ⟨*tokens*⟩ are initialized empty.

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`.

```
                "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=0pt and postbreak are used. }
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{␣'visiblespaces=true'␣implies␣'breakautoindent
  =false'.␣}
```

```
\lstset{postbreak=\space\space,breakindent=0pt,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=0pt and postbreak are used. }
\end{lstlisting}

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

3.4 Literate programming

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

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

if (i≤0) i ← 1;                         if (i<=0) i := 1;
if (i≥0) i ← 0;                         if (i>=0) i := 0;
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.

new `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
```

4 Troubleshooting

The known bugs have already been described. Contact me if you encounter any other problems if they are not described below. For a bug report create a *short* file which demonstrates the problem. Please start from the minimal file in section 1.3. However, include the file itself and the created `.log` file in your bug report.

4.1 Accents and splitted listings

Marcin Kasperski reported a deep problem and Donald Arseneau let me understand it a bit. Thanks to both. But now a description and possible solutions. Let's say that a new section starts on the current page and the section name contains an accented character. In particular this could be an extended character if you use inputenc (since that package makes these characters active and makes appropriate definitions). Furthermore a listing should start on the same page and end on the next page. Then you probably get an "undefined control sequence" error if you

try to make a `\tableofcontents`. The control sequence comes from the accent command and the character to be accented: A space is missing in between. The accents `\‘`, `\’`, `\^`, `\"`, etc. will work since there need not to be a space between command and accented character. But `\c`, `\d`, `\k`, etc. won't work: If you write `\c o`, the `.toc` file will show `\co` and that's usually undefined. The problem is neither limited to accents nor to `\tableofcontents`. Try `\index` and/or `\oe`, `\o`, `\L`, and so on.

If you type in the accents directly (accents for simplicity), you can make braces around the character to be accented, e.g. write `\c{o}` instead of `\c o` or write `{\oe}` instead of `\oe`. Then missing spaces don't hurt. This solution was proposed by Heiko Oberdiek.

If you use extended characters realized with inputenc, you can't go that way. I see two possibilities if you absolutely want to keep the inputenc package: (i) insert appropriate braces in `.def` files (but note the copyright and modification notices in these files!) or better (ii) let the listings package define a work-around.[3] Alternatively you could use fontenc. Marcin Kasperski solved the problem by using the web2c TEX character translation capability and used `%& -translate-file=`⟨*file*⟩ as the very first line of the document.

4.2 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. 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: Write

```
\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, folks!

5 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.

[3] The work-around would define a local version of the output routine: At the beginning we switch back to TEX's original catcodes and at the end we activate the listings' ones again.

There will definitely a `savemem` option. If used, it reduces the required TEX memory (and sometimes even runtime).

The procnames and index aspects are still unsatisfactory. For example, 'procnames' marks (and indexes) only the function definitions so far, but it would be possible to mark also the following function calls: Write another 'keyword class' which is empty at the very beginning (and can be reset with a key); each function definition appends a 'keyword' which will appear in 'procnamestyle'.

Torben Hoffmann stated the idea of pre-compiled listings. But if the package reads a listing to compare it with the pre-compiled version, all 'escape features' get lost (since the package can switch to its active characters, but not its active characters back to TEX's catcodes). Thus, pre-compiled listings are possibly never implemented.

From Rolf Niepraschk comes one more interesting idea, namely that two (or more) language definitions could be active at the same time. This would be useful if a makefile also contains parts of a shell-script language, for example.