The Listings Package

Copyright 1996–2004, Carsten Heinz
Copyright 2006–2007, Brooks Moses
Copyright 2013–, Jobst Hoffmann
Maintainer: Jobst Hoffmann* <j.hoffmann(at)fh-aachen.de>

2019/09/10 Version 1.8c

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 \verb 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.

User’s guide

1 Getting started
1.1 A minimal file
1.2 Typesetting listings
1.3 Figure out the appearance
1.4 Seduce to use
1.5 Alternatives
2 The next steps
2.1 Software license
2.2 Package loading
2.3 The key=value interface
2.4 Programming languages
2.4.1 Preferences
2.5 Special characters
2.6 Line numbers
2.7 Layout elements
2.8 Emphasize identifiers
2.9 Indexing
2.10 Fixed and flexible columns
3 Advanced techniques
3.1 Style definitions
3.2 Language definitions
3.3 Delimiters
3.4 Closing and credits

Reference guide

*Jobst Hoffmann became the maintainer of the listings package in 2013; see the Preface for details.

4 Main reference
4.1 How to read the reference
4.2 Typesetting listings
4.3 Options
4.3.1 Searching for files
4.3.2 Space and placement
4.3.3 The printed range
4.3.4 Languages and styles
4.3.5 Figure out the appearance
4.3.6 Getting all characters right
4.3.7 Line numbers
4.3.8 Captions
4.3.9 Margins and line shape
4.3.10 Frames
4.3.11 Indexing
4.3.12 Column alignment
4.3.13 Escaping to \LaTeX
4.4 Interface to fancyvrb
4.5 Environments
4.6 Short Inline Listing Commands
4.7 Language definitions
4.8 Installation

5 Experimental features
5.1 Listings inside arguments
5.2 † Export of identifiers
5.3 † Hyperlink references
5.4 Literate programming
5.5 LGrind definitions
5.6 † Automatic formatting
5.7 Arbitrary linerange markers
5.8 Multicolumn Listings
20 Interfaces to other programs 218
20.1 0.21 compatibility . . . . . . 218
20.2 fancyvrb . . . . . . . . . . . . 219
20.3 Omega support . . . . . . . . 222
20.4 LGrind . . . . . . . . . . . . . 223
20.5 hyperref . . . . . . . . . . . . 226
21 Epilogue 227
22 History 228

Index 230
Preface

Transition of package maintenance  The \TeX{} world lost contact with Carsten Heinz in late 2004, shortly after he released version 1.3b of the \texttt{listings} package. After many attempts to reach him had failed, Hendri Adriaens took over maintenance of the package in accordance with the LPPL’s procedure for abandoned packages. He then passed the maintainership of the package to Brooks Moses, who had volunteered for the position while this procedure was going through. The result is known as \texttt{listings} version 1.4.

This release, version 1.5, is a minor maintenance release since I accepted maintainership of the package. I would like to thank Stephan Hennig who supported the Lua language definitions. He is the one who asked for the integration of a new language and gave the impetus to me to become the maintainer of this package.

News and changes  Version 1.5 is the fifth bugfix release. There are no changes in this version, but two extensions: support of modern Fortran (2003, 2008) and Lua.

Thanks  There are many people I have to thank for fruitful communication, posting their ideas, giving error reports, adding programming languages to \texttt{lstdrvrs.dtx}, and so on. Their names are listed in section \ref{sect:Thanks}.

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

1 Getting started

1.1 A minimal file

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

```latex
\documentclass{article}
\usepackage{listings}
% \begin{document}
\lstset{language=Pascal}
% % Insert Pascal examples here.
% \end{document}
```

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

→ Must I do that really? Yes and no. Some books about programming say this is good. What a mistake! Typing takes time—which is 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 inexperienced 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 correctly.

→ The example doesn’t work. Are the two packages listings and keyval installed on your system? Consult the administration tool of your \TeX distribution, your system administrator, the local \TeX and \LaTeX guides, a \TeX FAQ, and section 4.8—in that order. If you’ve checked all these sources and are still helpless, you might want to write a post to a \TeX newsgroup like comp.text.tex.

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

1.2 Typesetting listings

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

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

**Code snippets** The well-known \LaTeX command \verb typesets code snippets verbatim. The new command \lstinline pretty-prints the code, for example \verb var i:integer;\ is typeset by \lstinline!var i:integer;!’. The exclamation marks delimit the code and can be replaced by any character not in the code; \lstinline$var i:integer;$ gives the same result.
**Displayed code** The `lstlisting` environment typesets the enclosed source code. Like most examples, the following one shows verbatim \LaTeX\ code on the right and the result on the left. You might take the right-hand side, put it into the minimal file, and run it through \LaTeX.

```
for i := maxint to 0 do begin
{ do nothing }
end;
Write('Case insensitive ');
Write('Pascal keywords. ');
```

It can’t be easier.

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

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

```
\begin{lstlisting}[firstline=2, lastline=5]
for i := maxint to 0 do begin
{ do nothing }
end;
Write('Case insensitive ');
Write('Pascal keywords. ');
\end{lstlisting}
```

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

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

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

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

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

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

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

1.3 Figure out the appearance

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

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

\begin{lstlisting}
for i:=maxint to 0 do
begin
  \{ do nothing \}
end;
Write('Case insensitive ');
Write('Pascal keywords.');\end{lstlisting}

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

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

→ 'basicstyle=\small’ looks fine, but comments look really bad with 'commentstyle=\tiny’
  and empty basic style, say. Don’t use different font sizes in a single listing.
→ But I really want it! No, you don’t.

Warning You should be very careful with striking styles; the recent example
is rather moderate—it can get horrible. Always use decent highlighting. Unfortunately it is difficult to give more recommendations since they depend on the
type of document you’re creating. Slides or other presentations often require more
striking styles than books, for example. In the end, it’s you who have to find the
golden mean!
Listing 1: A floating example

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

\begin{lstlisting}
Write('Case\_insensitive');
WriteE('Pascal\_keywords.');
\end{lstlisting}

1.4 Seduce to use

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

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

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

\begin{lstlisting}
for i:=maxint to 0 do
begin
  { do nothing }
end;
Write('Case insensitive');
WriteE('Pascal keywords.');
\end{lstlisting}

→ I can’t get rid of line numbers in subsequent listings. ‘numbers=none’ turns them off.

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

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

**Floating listings** Displayed listings may float:

\begin{lstlisting}[float,caption=A floating example]
for i:=maxint to 0 do
begin
  { do nothing }
end;
Write('Case insensitive');
WriteE('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.
\begin{lstlisting}
if (i<=0) then i := 1;
if (i>=0) then i := 0;
if (i<>0) then i := 0;
\end{lstlisting}

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

1.5 Alternatives

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

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

This package is certainly not the final utility for typesetting source code. Other programs do their job very well, if you are not satisfied with listings. Some are independent of \LaTeX, others come as separate program plus \LaTeX package, and others are packages which don’t pretty-print the source code. The second type includes 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 tools here, which could also be alternatives. 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.

\texttt{a2ps} started as ‘ASCII to PostScript’ converter, but today you can invoke the program with \texttt{--pretty-print=(language)} option. If your favourite programming language is not already supported, you can write your own so-called style sheet. You can request line numbers, borders, headers, multiple pages per sheet, and many more. You can even print symbols like $\forall$ or $\alpha$ instead of their verbose forms.

If you just want program listings and not a document with some listings, this is the best choice.

\texttt{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 \texttt{-e} option (and file names), and run the created file through \LaTeX. You should get a result similar to the left-hand side:
If you use `%( and %)` instead of `%{ and %}`, you get a code snippet instead of a displayed listing. Moreover you can get line numbers to the left or right, use arbitrary \LaTeX{} code in the source code, print symbols instead of verbose names, make font setup, and more. You will (have to) like it (if you don’t like listings).

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

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

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

\texttt{SL\LaTeX{}} is a pretty-printing Scheme program (which 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.

\texttt{tiny}\texttt{c2ltx} is a C/C++/Java to \LaTeX{} converter based on \texttt{cvt2ltx} (or the other way round?). It supports line numbers, block comments, \LaTeX{} code in/as comments, and smart line breaking. Font selection and tabulators are hard-coded, i.e. you have to rebuild the program if you want to change the appearance.

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

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

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

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

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

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

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

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

fancyverb is, roughly speaking, a superset 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 fvrβ-ex builds on fancyverb and defines environments to present examples similar to the ones in this guide.

2 The next steps

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

2.1 Software license

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

Copyright The listings package is copyright 1996–2004 Carsten Heinz, and copyright 2006 Brooks Moses. The drivers are copyright any individual author listed in the driver files.

Distribution and modification The listings package and its drivers may be distributed and/or modified under the conditions of the LaTeX Project Public License, either version 1.3 of this license or (at your option) any later version. The latest version of this license is in http://www.latex-project.org/lppl.txt and version 1.3c or later is part of all distributions of LaTeX version 2003/12/01 or later.

Contacts Read section 6 Troubleshooting on how to submit a bug report. Send all other comments, ideas, and additional programming languages to j.hoffmann(at)fh-aachen.de using listings as part of the subject.
2.2 Package loading

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

→ Where is a list of all of the options? In the developer’s guide since they were introduced to debug the package more easily. Read section 8 on how to get that guide.

0.21

invokes a compatibility mode for compiling documents written for listings version 0.21.

draft

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

final

Overwrites a global draft option.

savemem

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

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

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

\texttt{\lstloadlanguages\{(comma separated list of languages)\}}

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

Table 1 on page 14 shows all defined languages and their dialects.

2.3 The key=value interface

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

The command \texttt{\lstset} gets a comma separated list of “key=value” pairs. The first list with more than a single entry is on page 6: \texttt{firstline=2, lastline=5}. 12
So I can write \lstset{firstline=2,lastline=5} once for all? No. ‘firstline’ and ‘lastline’ belong to a small set of keys which are only 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 unless you specify at least an empty optional argument as in ‘\lstinline[\{\}][var i:integer;\]’. If you forget this, you will either get a “runaway argument” error from TeX, or an error message from the keyval package.

2.4 Programming languages

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

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

How can I define default dialects? Check section 4.3.4 for ‘defaultdialect’.

I have C code mixed with assembler lines. Can listings pretty-print such source code, i.e. highlight keywords and comments of both languages? ‘\alsolanguage=\{\langle dialect\rangle\}\langle language\rangle’ selects a language additionally to the active one. So you only have to write a language definition for your assembler dialect, which doesn’t interfere with the definition of C, say. Moreover you might want to use the key ‘classoffset’ described in section 4.3.4.

How can I define my own language? This is discussed in section 4.7. And if you think that other people could benefit by your definition, you might want to send it to the address in section 2.1. Then it will be published under the \LaTeX{} Project Public License.

Note that the arguments \langle language\rangle and \langle dialect\rangle are case insensitive and that spaces have no effect.

There is at least one language (VDM, Vienna Development Language, \url{http://www.vdmportal.org}) which is not directly supported by the listings package. It needs a package for its own: vdmlisting. On the other hand vdmlisting uses the listings package and so it should be mentioned in this context.

2.4.1 Preferences

Sometimes authors of language support provide their own configuration preferences. These may come either from their personal experience or from the
Table 1: Predefined languages. Note that some definitions are preliminary, for example HTML and XML. Each underlined dialect is the default dialect.

<table>
<thead>
<tr>
<th>Language</th>
<th>Version</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAP</td>
<td>R/2 4.3, R/2 5.0, R/3 3.1, R/3 4.6C, R/3 6.10</td>
<td></td>
</tr>
<tr>
<td>ACM</td>
<td>ACMscript</td>
<td></td>
</tr>
<tr>
<td>ACMscript</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSL</td>
<td>Ada (2005, 83, 95)</td>
<td></td>
</tr>
<tr>
<td>Algol (60, 68)</td>
<td>Ant</td>
<td></td>
</tr>
<tr>
<td>Assembler (Motorola68k, x86masm)</td>
<td>Awk (gnu, POSIX)</td>
<td></td>
</tr>
<tr>
<td>bash</td>
<td>Basic (Visual)</td>
<td></td>
</tr>
<tr>
<td>C (ANSI, Handel, Objective, Sharp)</td>
<td>Caml (light, Objective)</td>
<td></td>
</tr>
<tr>
<td>C++ (11, ANSI, GNU, ISO, Visual)</td>
<td>Clean</td>
<td></td>
</tr>
<tr>
<td>CIL</td>
<td>Comal 80</td>
<td></td>
</tr>
<tr>
<td>command.com (WinXP)</td>
<td>csh</td>
<td></td>
</tr>
<tr>
<td>Eiffel</td>
<td>Delphi</td>
<td></td>
</tr>
<tr>
<td>elisp</td>
<td>erlang</td>
<td></td>
</tr>
<tr>
<td>Euphoria</td>
<td>Fortran (03, 08, 77, 90, 95)</td>
<td></td>
</tr>
<tr>
<td>GAP</td>
<td>GCL</td>
<td></td>
</tr>
<tr>
<td>Gnuplot</td>
<td>Go</td>
<td></td>
</tr>
<tr>
<td>hansl</td>
<td>Haskell</td>
<td></td>
</tr>
<tr>
<td>HTML</td>
<td>IDL (empty, CORBA)</td>
<td></td>
</tr>
<tr>
<td>inform</td>
<td>Java (empty, AspectJ)</td>
<td></td>
</tr>
<tr>
<td>JVMIS</td>
<td>ksh</td>
<td></td>
</tr>
<tr>
<td>Lingo</td>
<td>Lisp (empty, Auto)</td>
<td></td>
</tr>
<tr>
<td>LLVM</td>
<td>Logo</td>
<td></td>
</tr>
<tr>
<td>Lua (5.0, 5.1, 5.2, 5.3)</td>
<td>make (empty, gnu)</td>
<td></td>
</tr>
<tr>
<td>Mathematica (1.0, 11.0, 3.0, 5.2)</td>
<td>Matlab</td>
<td></td>
</tr>
<tr>
<td>Mercury</td>
<td>MetaPost</td>
<td></td>
</tr>
<tr>
<td>Miranda</td>
<td>Mizar</td>
<td></td>
</tr>
<tr>
<td>ML</td>
<td>Modula-2</td>
<td></td>
</tr>
<tr>
<td>MuPAD</td>
<td>NASTRAN</td>
<td></td>
</tr>
<tr>
<td>Oberon-2</td>
<td>OCL (decorative, OMG)</td>
<td></td>
</tr>
<tr>
<td>Octave</td>
<td>OORexx</td>
<td></td>
</tr>
<tr>
<td>Oz</td>
<td>Pascal (Borland6, Standard, XSC)</td>
<td></td>
</tr>
<tr>
<td>Perl</td>
<td>PHP</td>
<td></td>
</tr>
<tr>
<td>PL/I</td>
<td>Plasm</td>
<td></td>
</tr>
<tr>
<td>PostScript</td>
<td>POV</td>
<td></td>
</tr>
<tr>
<td>Prolog</td>
<td>Promela</td>
<td></td>
</tr>
<tr>
<td>PSTricks</td>
<td>Python</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>Reduce</td>
<td></td>
</tr>
<tr>
<td>Rexx (empty, VM/XA)</td>
<td>RSL</td>
<td></td>
</tr>
<tr>
<td>Ruby</td>
<td>S (empty, PLUS)</td>
<td></td>
</tr>
<tr>
<td>SAS</td>
<td>Scala</td>
<td></td>
</tr>
<tr>
<td>Scalab</td>
<td>sh</td>
<td></td>
</tr>
<tr>
<td>SHEX/XL</td>
<td>Simula (67, CII, DEC, IBM)</td>
<td></td>
</tr>
<tr>
<td>SPARQL</td>
<td>SQL</td>
<td></td>
</tr>
<tr>
<td>Swift</td>
<td>tcl (empty, tk)</td>
<td></td>
</tr>
<tr>
<td>TeX (AlaTeX, common, LaTeX, plain, primitive)</td>
<td>Verilog</td>
<td></td>
</tr>
<tr>
<td>VBScript</td>
<td>Verilog</td>
<td></td>
</tr>
<tr>
<td>VHDL (empty, AMS)</td>
<td>VRML (97)</td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td>XSLT</td>
<td></td>
</tr>
</tbody>
</table>
settings in an IDE and can be defined as a listings style. From version 1.5b of the listings package on these styles are provided as files with the name listings-⟨language⟩.prf, ⟨language⟩ is the name of the supported programming language in lowercase letters.

So if an user of the listings package wants to use these preferences, she/he can say for example when using Python

\input{listings-python.prf}

at the end of her/his listings.cfg configuration file as long as the file listings-python.prf resides in the \TeX{} search path. Of course that file can be changed according to the user’s preferences.

At the moment there are five such preferences files:

1. listings-acm.prf
2. listings-bash.prf
3. listings-fortran.prf
4. listings-lua.prf
5. listings-python.prf

All contributors are invited to supply more personal preferences.

2.5 Special characters

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

\begin{lstlisting}
  123456789
  { one tabulator }
  { two tabs }
  123 { 123 + two tabs }
\end{lstlisting}

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

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

\begin{lstlisting}
\lstset{showspaces=true, showtabs=true, tab=\rightarrowfill}
  \begin{lstlisting}
  for i:=maxint to 0 do
  begin
  { do nothing }
  end;
\end{lstlisting}
\end{lstlisting}
If you request `showspaces` but no `showtabs`, tabulators are converted to visible spaces. The default definition of `tab` produces a ‘wide visible space’ \ldots. So you might want to use \textbackslash \texttt{\{to$}, \textbackslash \texttt{dash}$ or something else instead.

\begin{itemize}
\item 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!
\end{itemize}

\begin{itemize}
\item To make the \LaTeX{} code more readable, I indent the environments’ program listings. How can I remove that indentation in the output? Read ‘How to gobble characters’ in section 8.
\end{itemize}

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

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

\begin{itemize}
\item I have problems using `inputenc` together with listings. This could be a compatibility problem. Make a bug report as described in section 6 Troubleshooting.
\end{itemize}

The extended characters don’t cover Arabic, Chinese, Hebrew, Japanese, and so on—specifically, any encoding which uses multiple bytes per character.

Thus, if you use the a package that supports multibyte characters, such as the CJK or `ucs` packages for Chinese and UTF-8 characters, you must avoid letting listings process the extended characters. It is generally best to also specify `extendedchars=false` to avoid having listings get entangled in the other package’s extended-character treatment.

If you do have a listing contained within a CJK environment, and want to have CJK characters inside the listing, you can place them within a comment that escapes to \LaTeX{}—see section 4.3.13 for how to do that. (If the listing is not inside a CJK environment, you can simply put a small CJK environment within the escaped-to-\LaTeX{}portion of the comment.)

Similarly, if you are using UTF-8 extended characters in a listing, they must be placed within an escape to \LaTeX{}.

Also, section 8 has a few details on how to work with extended characters in the context of \LaTeX{}.

### 2.6 Line numbers

You already know the keys `numbers`, `numberstyle`, `stepnumber`, and `numbersep` from section 1.4. Here now we deal with continued listings. You have two options to get consistent line numbering across listings.
for \( i := \text{maxint} \) to 0 do \\
begin \{ \text{do nothing } \} \\
end;

And we continue the listing:

\begin{lstlisting}[firstnumber=last]
Write('Case insensitive '); \\
Write('Pascal keywords.'); \\
\end{lstlisting}

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

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

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

And we continue the listing:

\begin{lstlisting}[name=Test]
Write('Case insensitive '); \\
Write('Pascal keywords.'); \\
\end{lstlisting}

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

→ Okay. And how can I get decreasing line numbers? Sorry, what? Decreasing line numbers as on page 35. May I suggest to demonstrate your individuality by other means? If you differ, you should try a negative \texttt{stepnumber} (together with \texttt{firstnumber}).

Read section 8 on how to reference line numbers.

2.7 Layout elements

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

\textbf{Vertical space} The keys \texttt{aboveskip} and \texttt{belowskip} control the vertical space above and below displayed listings. Both keys get a dimension or skip as value and are initialized to \texttt{\medskipamount}.
Frames  The key frame takes the verbose values none, leftline, topline, bottomline, lines (top and bottom), single for single frames, or shadowbox.

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

\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 4.3.10 heavily. And do this step by step! For example, begin with ’framerule=10mm’. If the rules are misaligned by the same (small) amount as before, the problem does not come from the rule width. So continue with the next parameter. Also, Adobe Acrobat sometimes has single-pixel rounding errors which can cause small misalignments at the corners when PDF files are displayed on screen; these are unfortunately normal.

Alternatively you can control the rules at the top, right, bottom, and left directly by using the four initial letters for single rules and their upper case versions for double rules.

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

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

\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.

→ Don’t use frames all the time, and 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 \TeX’s \texttt{\textbackslash topfigrule} and \texttt{\textbackslash botfigrule}. For example, you could write \texttt{\renewcommand\textbackslash topfigrule{\textbackslash hrule\textbackslash kern=-0.4\textbackslash pt\textbackslash relax}} and make the same definition for \texttt{\textbackslash botfigrule}.
Captions  Now we come to \texttt{caption} and \texttt{label}. You might guess (correctly) that they can be used in the same manner as \LaTeX{}'s \texttt{caption} and \texttt{label} commands, although here it is also possible to have a caption regardless of whether or not the listing is in a float:

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

Listing 2: Useless code

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

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

If you don’t want the label \texttt{Listing plus number}, you should use \texttt{title}:

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

‘Caption without label

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

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

Colours  One more element. You need the \texttt{color} package and can then request coloured background via \texttt{\textcolor{\texttt{\langle color command\rangle}}}{backgroundcolor=\texttt{(color command)}}.

→ Great! I love colours. Fine, yes, really. And I like to remind you of the warning about striking styles on page 7.

\lstset{backgroundcolor=\textcolor{\texttt{\color{yellow}}}}
for \(i := \text{maxint} \) to 0 do
begin
  \( j := \text{square} ( \text{root}(i)) \);
end;

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

2.8 Emphasize identifiers

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

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

\begin{lstlisting}
\lstset{emph={square,root},emphstyle=\underbar}

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

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

Both keys have an optional \texttt{(class number)} argument for multiple identifier lists:

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

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

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

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

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

2.9 Indexing

Indexing is just like emphasizing identifiers—I mean the usage:

\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.

→ Why is the ‘index’ key 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=\indexkeywords’ might meet your purpose. This becomes easier if you want to create multiple indexes with the index package. If you have defined appropriate new indexes, it is possible to write ‘indexstyle=\index{keywords}’, for example.

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

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

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

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

21
2.10 Fixed and flexible columns

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

\[
\begin{array}{cccc}
\text{columns= } & \text{fixed} & \text{flexible} & \text{fullflexible} \\
& (\text{at 0.6em}) & (\text{at 0.45em}) & (\text{at 0.45em}) \\
\text{WOMEN are } & \text{WOMEN are } & \text{WOMEN are } & \text{WOMEN are} \\
\text{MEN} & \text{MEN} & \text{MEN} & \text{MEN} \\
\text{WOMEN are } & \text{WOMEN are } & \text{WOMEN are } & \text{WOMEN are} \\
\text{better MEN} & \text{better MEN} & \text{better MEN} & \text{better MEN} \\
\end{array}
\]

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

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

\[
\text{if } x - y \text{ then write (‘align’)};\quad \text{else print (‘align’)};
\]

Now, the fixed format puts \( n \) characters into a box of width \( n \times \text{‘base width’} \), where the base width is 0.6em in the example. The format shrinks and stretches the space between the characters to make them fit the box. As shown in the example, some character strings look bad or worse, but the output is vertically aligned.

If you don’t need or like this, you should use a flexible format. All characters are typeset at their natural width. In particular, they never overlap. If a word requires more space than reserved, the rest of the line simply moves to the right. The difference between the three formats is that the full flexible format cares about nothing else, while the normal flexible and space-flexible formats try to fix the column alignment if a character string needs less space than ‘reserved’. The normal flexible format will insert make-up space to fix the alignment at spaces, before and after identifiers, and before and after sequences of other characters; the space-flexible format will only insert make-up space by stretching existing spaces. In the flexible example above, the two MENs are vertically aligned since some space has been inserted in the fourth line to fix the alignment. In the full flexible format, the two MENs are not aligned.

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

3.1 Style definitions

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

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

\begin{verbatim}
% \lstdefinestyle{numbers}
% {numbers=left, stepnumber=1, numberstyle=\tiny, numbersep=10pt}
% \lstdefinestyle{nonumbers}
% {numbers=none}
\end{verbatim}

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

\begin{itemize}
  \item You could even write \texttt{\lstdefinestyle{C++}{language=C++,style=numbers}}. Style and language names are independent of each other and so might coincide. Moreover it is possible to activate other styles.
  \item It's easy to crash the package using styles. Write \texttt{\lstdefinestyle{crash}{style=crash}} and \texttt{\lstset{style=crash}}. \TeX{}'s capacity will exceed, sorry \cite{parameter stack size}. Only bad boys use such recursive calls, but only good girls use this package. Thus the problem is of minor interest.
\end{itemize}

3.2 Language definitions

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

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

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

\begin{itemize}
  \item I get a 'Missing = inserted for \texttt{\ifnum} error when I select my language. Did you forget the comma after 'keywords={...}'? And if you encounter unexpected characters after selecting a language (or style), you have probably forgotten a different comma or you have given to many arguments to a key, for example, \texttt{morecomment=[1]{\textbackslash--\textbackslash!}}.
\end{itemize}
So let’s turn to comments and strings. Each value starts with a *mandatory* 
[⟨type⟩] argument followed by a changing number of opening and closing delimiters. Note that each delimiter (pair) requires a key=value on its own, even if types are equal. Hence, you’ll need to insert `morestring=[b]'` if single quotes open and close string or character literals in the same way as double quotes do in the example.

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

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

\end{lstlisting}
```

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

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

### 3.3 Delimiters

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

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

\end{lstlisting}
```
As you can see, you have the choice between specifying the style explicitly by \texttt{LATEX} commands or implicitly by other style keys. But, you’re right, some implicitly defined styles have no separate keys, for example the second order keyword style. Here—and never with the number 1—you just append the order to the base key: \texttt{keywordstyle2}.

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

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

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

\begin{itemize}
\item Please remember that both ‘extra’ comments must be defined after the normal comment, since the delimiter ‘*/’ is a substring of ‘/*+’ and ‘/*–’.
\item I have another question. Is ‘language=⟨different language⟩’ the only way to remove such additional delimiters? Call \texttt{deletecomment} and/or \texttt{deletestring} with the same arguments to remove the delimiters (but you don’t need to provide the optional style argument).
\end{itemize}

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

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

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

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

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

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

Moreover, you can force the styles to be applied cumulatively.
Look carefully at the output and note the differences. The second `begin` is not printed in bold typewriter type since standard \LaTeX{} has no such font.

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

### 3.4 Closing and credits

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

→ There is one question ‘you’ haven’t asked all the last pages: who is to blame. Carsten Heinz wrote the guides, coded the `listings` package and wrote some language drivers. Brooks Moses took over the maintaining for several years, Jobst Hoffmann currently maintains the package. Other people defined more languages or contributed their ideas; many others made bug reports, but only the first bug finder is listed. Special thanks go to (alphabetical order)

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

Moreover we wish to thank

4 Main reference

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

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

4.1 How to read the reference

Commands, keys and environments are presented as follows.

\texttt{\textit{command}, \textit{environment} or \textit{key} with (parameters) } \textbf{default}

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

If present, the label in the left margin provides extra information: ‘\texttt{addon}’ indicates additionally introduced functionality, ‘\texttt{changed}’ a modified key, ‘\texttt{data}’ a command just containing data (which is therefore adjustable via \texttt{\renewcommand}), and so on. Some keys and functionality are ‘\texttt{bug}’-marked or with a ‡-sign. These features might change in future or could be removed, so use them with care.
If there is verbatim text touching the right margin, it is the predefined value. Note that some keys default to this value every listing, namely the keys which can be used on individual listings only.

Regarding the parameters, please keep in mind the following:

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

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

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

4. A vertical rule indicates an alternative, e.g. \begin{itemize}[true\ false] allows either true or false as arguments.

5. If you want to enter one of the special characters {}#\ everything must be escaped with a backslash. This means that you must write \} for the single character ‘right brace’—but of course not for the closing parameter character.

### 4.2 Typesetting listings

\lstset{⟨key=value list⟩}

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

\lstinline![⟨key=value list⟩]{⟨source code⟩}{⟨same character⟩}

works like \verb but respects the active language and style. These listings use flexible columns unless requested differently in the optional argument, and do not support frames or background colors. You can write ‘\lstinline!var i:integer;!’ and get ‘var i:integer;!’. Since the command first looks ahead for an optional argument, you must provide at least an empty one if you want to use \ as ⟨character⟩.

† An experimental implementation has been done to support the syntax \lstinline[⟨key=value list⟩]{⟨source code⟩}. Try it if you want and report success and failure. A known limitation is that inside another argument the last source code token must not be an explicit space token—and, of course, using a listing inside another argument is itself experimental, see section 5.1. Another limitation is that this feature can’t be used in cells of a tabular environment. See section 7.1 for a workaround.

See also section 4.6 for commands to create short analogs for the \lstinline command.
\begin{lstlisting}[(key=value list)]
\end{lstlisting}

typesets the code in between as a displayed listing.

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

\begin{lstlisting}[(key=value list)]{(file name)}
\end{lstlisting}

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

4.3 Options

The following sections describe all the keys that can be used to influence the
appearance of the listing.

4.3.1 Searching for files

\begin{verbatim}
inputpath=\langle path \rangle
\end{verbatim}

defines the path, where the file given by \langle file name \rangle resides.

\texttt{inputpath} overrules the \texttt{TEXINPUTS} environment variable, which means that
a file residing on one of the paths given by \texttt{TEXINPUTS} isn’t found anymore,
if \langle path \rangle isn’t part of \texttt{TEXINPUTS}.

\texttt{inputpath} set as option of \texttt{\lstinputlisting} overrules the value set by
\texttt{\lstset}.

4.3.2 Space and placement

\begin{verbatim}
float=\langle subset of tbph \rangle or float \hspace{1em} floatplacement
\end{verbatim}

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

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

\begin{verbatim}
floatplacement=\langle place specifiers \rangle \hspace{1em} \texttt{tbp}
\end{verbatim}

is used as place specifier if \texttt{float} is used without value.

\begin{verbatim}
aboveskip=\langle dimension \rangle \hspace{1em} \texttt{\medskipamount}
belowskip=\langle dimension \rangle \hspace{1em} \texttt{\medskipamount}
\end{verbatim}

define the space above and below displayed listings.

\begin{verbatim}
\texttt{/ lineskip=\langle dimension \rangle} \hspace{1em} \texttt{Opt}
\end{verbatim}

specifies additional space between lines in listings.

\begin{verbatim}
\texttt{/ boxpos=\langle b|c|t \rangle} \hspace{1em} \texttt{c}
\end{verbatim}

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

\texttt{print=}\langle\texttt{true|false}\rangle \text{ or } \texttt{print}

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

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

\texttt{firstline=}\langle\texttt{number}\rangle 1
\texttt{lastline=}\langle\texttt{number}\rangle 9999999

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

\texttt{linerange=\{\langle\texttt{first1}\rangle-\langle\texttt{last1}\rangle, \langle\texttt{first2}\rangle-\langle\texttt{last2}\rangle, \text{ and so on}\}}

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

\texttt{showlines=}\langle\texttt{true|false}\rangle \text{ or } \texttt{showlines}

false

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

\texttt{emptylines=}\star\langle\texttt{number}\rangle

sets the maximum of empty lines allowed. If there is a block of more than \langle\texttt{number}\rangle empty lines, only \langle\texttt{number}\rangle ones are printed. Without the optional star, line numbers can be disturbed when blank lines are omitted; with the star, the lines keep their original numbers.

\texttt{gobble=}\langle\texttt{number}\rangle 0

gobbles \langle\texttt{number}\rangle characters at the beginning of each \texttt{environment} code line. This key has no effect on \texttt{\lstinline} or \texttt{\lstinputlisting}.

Tabulators expand to \texttt{tabsize} spaces before they are gobbled. Code lines with fewer than \texttt{gobble} characters are considered empty. Never indent the end of environment by more characters.

4.3.4 Languages and styles

Please note that the arguments \langle\texttt{language}\rangle, \langle\texttt{dialect}\rangle, and \langle\texttt{style name}\rangle are case insensitive and that spaces have no effect.

\texttt{style=}\langle\texttt{style name}\rangle \{\}

activates the key=value list stored with \texttt{\lstdefinestyle}.

\texttt{\lstdefinestyle}\{\langle\texttt{style name}\rangle\}{\langle\texttt{key=value list}\rangle}

stores the key=value list.

\texttt{language=}\{\langle\texttt{dialect}\rangle\}\{\langle\texttt{language}\rangle\} \{\}

activates a (dialect of a) programming language. The ‘empty’ default language detects no keywords, no comments, no strings, and so on; it may
be useful for typesetting plain text. If \textit{(dialect)} is not specified, the package chooses the default dialect, or the empty dialect if there is no default dialect. Table 1 on page 14 lists all languages and dialects provided by \texttt{lstdrvrs.dtx}. The predefined default dialects are underlined.

\texttt{alsolanguage=[\langle \textit{dialect} \rangle ] \langle \textit{language} \rangle}

activates a (dialect of a) programming language in addition to the current active one. Note that some language definitions interfere with each other and are plainly incompatible; for instance, if one is case sensitive and the other is not.

Take a look at the \textit{classoffset} key in section 4.3.5 if you want to highlight the keywords of the languages differently.

\texttt{defaultdialect=[\langle \textit{dialect} \rangle ] \langle \textit{language} \rangle}

defines \textit{(dialect)} as default dialect for \textit{(language)}. If you have defined a default dialect other than empty, for example \texttt{defaultdialect=[iama]fool}, you can’t select the empty dialect, even not with \texttt{language=[\]}\texttt{fool}.

Finally, here’s a small list of language-specific keys.

\texttt{optional printpod=(true|false)}

prints or drops PODs in Perl.

\texttt{renamed,optional usekeywordsintag=(true|false)}

The package either use the first order keywords in tags or prints all identifiers inside \langle $ \rangle in keyword style.

\texttt{optional tagstyle=(style)}

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

\texttt{optional markfirstintag=(style)}

prints the first name in tags with keyword style.

\texttt{optional makemacrouse=(true|false)}

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

\subsection{4.3.5 Figure out the appearance}

\texttt{basicstyle=(basic style)}

is selected at the beginning of each listing. You could use \texttt{\footnotesize}, \texttt{\small}, \texttt{\ttfamily}, or something like that. The last token of \textit{(basic style)} must not read any following characters.

\texttt{identifierstyle=(style)}

\texttt{commentstyle=(style)}

\texttt{\itshape}
stringstyle=⟨style⟩

determines the style for non-keywords, comments, and strings. The last
token can be an one-parameter command like \textbf or \underline.

addon keywordstyle=⟨{number}⟩[∗]{style}
\bfseries

is used to print keywords. The optional ⟨{number}⟩ argument is the class
number to which the style should be applied.
Add-on: If you use the optional star after the (optional) class number, the
keywords are printed uppercase—even if a language is case sensitive and
defines lowercase keywords only. Maybe there should also be an option for
lowercase keywords . . .

deprecated ndkeywordstyle=⟨style⟩

is equivalent to keywordstyle=2{style}.

classoffset=⟨{number}⟩

0

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

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

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

addon,bug,optional texcsstyle=[∗]{⟨class number⟩}{⟨style⟩}

optional directivestyle=⟨{style}⟩

keywordstyle
determine the style of \TeX control sequences and directives. Note that these
keys are present only if you’ve chosen an appropriate language.
The optional star of texcsstyle also highlights the backslash in front of the
control sequence name. Note that this option is set for all texcs lists.

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

emph=[⟨{number}⟩]{⟨identifier list⟩}

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

deleteemph=[⟨{number}⟩]{⟨identifier list⟩}
emphstyle=\{(number)\}\{(style)\}
respectively define, add or remove the \langle identifier list \rangle from \textit{emphasize class} \langle number \rangle, or define the style for that class. If you don’t give an optional argument, the package assumes \langle number \rangle = 1.

These keys are described more detailed in section 2.8.

delim=[*][\langle type \rangle][\langle style \rangle][\langle delimiter(s) \rangle]
moredelim=[*][\langle type \rangle][\langle style \rangle][\langle delimiter(s) \rangle]
deletedelim=[*][\langle type \rangle][\langle delimiter(s) \rangle]
define, add, or remove user supplied delimiters. (Note that this does not affect strings or comments.)

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

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

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

4.3.6 Getting all characters right

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

inputencoding=\{encoding\} {}
determines the input encoding. The usage of this key requires the inputenc package; nothing happens if it’s not loaded.

upquote=(true|false) \false
determines whether the left and right quote are printed ’ ’ or ``. This key requires the textcomp package if true.

tabsize=\langle number \rangle \8
sets tabulator stops at columns \langle number \rangle + 1, 2:\langle number \rangle + 1, 3:\langle number \rangle + 1, and so on. Each tabulator in a listing moves the current column to the next tabulator stop.

showtabs=(true|false) \false
make tabulators visible or invisible. A visible tabulator looks like \underline{underline}, but that can be changed. If you choose invisible tabulators but visible spaces, tabulators are converted to an appropriate number of spaces.
tab=⟨tokens⟩

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

showspaces=⟨true|false⟩

false

lets all blank spaces appear \ or as blank spaces.

showstringspaces=⟨true|false⟩

true

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

formfeed=⟨tokens⟩

\bigbreak

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

4.3.7 Line numbers

numbers=⟨none|left|right⟩

none

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

stepnumber=⟨number⟩

1

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

numberfirstline=⟨true|false⟩

false

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

numberstyle=⟨style⟩

{}\}

determines the font and size of the numbers.

numbersep=⟨dimension⟩

10pt

is the distance between number and listing.

numberblanklines=⟨true|false⟩

true

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

firstnumber=⟨auto|last|(number)⟩

auto

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

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

name=⟨name⟩

names a listing. Displayed environment-listings with the same name share a line counter if firstnumber=auto is in effect.
data \thelstnumber \arabic{lstnumber}

prints the lines' numbers.

We show an example on how to redefine \thelstnumber. But if you test it, you
won't get the result shown on the left.
\renewcommand*{\thelstnumber}{\oldstylenums{\the\value{lstnumber}}}
\begin{lstlisting}[numbers=left,
firstnumber=753]
begin { empty lines }
\end{lstlisting}

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

→ The example shows a sequence \(n, n+1, \ldots, n+7\) of 8 three-digit figures such that the
sequence contains each digit 0, 1, \ldots, 9. But 8 is not minimal with that property. Find the
minimal number and prove that it is minimal. How many minimal sequences do exist?

Now look at the generalized problem: Let \(k \in \{1, \ldots, 10\}\) be given. Find the minimal number
\(m \in \{1, \ldots, 10\}\) such that there is a sequence \(n, n+1, \ldots, n+m-1\) of \(m\) \(k\)-digit figures
which contains each digit \(0, \ldots, 9\). Prove that the number is minimal. How many minimal
sequences do exist?

If you solve this problem with a computer, write a \TeX\ program!

4.3.8 Captions

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

\texttt{\lstlistoflistings}

The header name for the list of listings.
\lstlistingname \lstname

The caption label for listings.

\lstlistingnamestyle {}

customizes the style of the caption label for program listings in a simple way, something like \texttt{\small, \textbf{series}} or a combination of several commands. If there is a need for a complete customization of the label (justification, fonts, margins, ...), one should use the \texttt{caption} package by A. Sommerfeldt \cite{caption}.

\thelstlisting \arabic{lstlisting}

prints the running number of the caption.

\texttt{numberbychapter=⟨true|false⟩}

\texttt{true}
If true, and \texttt{\thechapter} exists, listings are numbered by chapter. Otherwise, they are numbered sequentially from the beginning of the document. This key can only be used before \texttt{\begin{document}}.

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

\texttt{captionpos=⟨subset of tb⟩}
\texttt{t}
specifies the positions of the caption: top and/or bottom of the listing.

\texttt{abovecaptionskip=⟨dimension⟩} \\smallskipamount \texttt{\smallskipamount}
\texttt{belowcaptionskip=⟨dimension⟩} \\smallskipamount \texttt{\smallskipamount}
is the vertical space respectively above or below each caption.

4.3.9 Margins and line shape

\texttt{linewidth=⟨dimension⟩} \\linewidth
defines the base line width for listings. The following three keys are taken into account additionally.

\texttt{xleftmargin=⟨dimension⟩} \texttt{Opt}
\texttt{xrightmargin=⟨dimension⟩} \texttt{Opt}
The dimensions are used as extra margins on the left and right. Line numbers and frames are both moved accordingly.

\texttt{resetmargins=⟨true|false⟩} \texttt{false}
If true, indentation from list environments like \texttt{enumerate} or \texttt{itemize} is reset, i.e. not used.

\texttt{breaklines=⟨true|false⟩ or breaklines} \texttt{false}
activates or deactivates automatic line breaking of long lines.
breakatwhitespace=\{true|false\} or breakatwhitespace false

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

prebreak=\{tokens\} {}

postbreak=\{tokens\} {}

\{tokens\} appear at the end of the current line respectively at the beginning of the next (broken part of the) line.

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

breakindent=\{dimension\} 20pt

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

breakautoindent=\{true|false\} or breakautoindent true

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

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

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

"A long string is broken!"
"Another long line."
\begin{lstlisting} { Now auto indention is off. } \end{lstlisting}
\begin{lstlisting}[breakautoindent=false] { Now auto indention is off. } \end{lstlisting}

4.3.10 Frames

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

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

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

frame=\{subset of trblTRBL\} {}

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

Note that frames usually reside outside the listing’s space.
framerecoard={t|f}{t|f}{t|f}{t|f}  \texttt{ffff}

The four letters designate the top right, bottom right, bottom left and top left corner. In this order, \texttt{t} makes the according corner round. If you use round corners, the rule width is controlled via \texttt{\thinlines} and \texttt{\thicklines}.

Note: The size of the quarter circles depends on \texttt{framesep} and is independent of the extra margins of a frame. The size is possibly adjusted to fit \LaTeX’s circle sizes.

\texttt{framesep=(dimension)}  \hspace{3pt}
\texttt{rulesep=(dimension)}  \hspace{2pt}

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

\texttt{framerule=(dimension)}  \hspace{0.4pt}
control the width of the rules.

\texttt{framexleftmargin=(dimension)}  \hspace{0pt}
\texttt{framexrightmargin=(dimension)}  \hspace{0pt}
\texttt{framextopmargin=(dimension)}  \hspace{0pt}
\texttt{framexbottommargin=(dimension)}  \hspace{0pt}

are the dimensions which are used additionally to \texttt{framesep} to make up the margin of a frame.

\texttt{backgroundcolor=(color command)}
\texttt{rulecolor=(color command)}
\texttt{fillcolor=(color command)}
\texttt{rulesepcolor=(color command)}

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

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

\texttt{\lstset{framexleftmargin=5mm, frame=shadowbox, rulesepcolor=\color{blue}}}  \begin{lstlisting}
[numbers=left]
for i:=maxint to 0 do
begin
  \{ do nothing \}
end;
\end{lstlisting}
\texttt{\begin{lstlisting}}
begin{lstlisting}[numbers=left]
for i:=\maxint to 0 do
begin
  \{ do nothing \}
end;
end{lstlisting}
\end{lstlisting}

Note here the use of \texttt{framexleftmargin} to include the line numbers inside the frame.

Do you want exotic frames? Try the following key if you want, for example,
\begin{lstlisting}
for i:=maxint to 0 do 
begin 
   \{ do nothing \}
end;
\end{lstlisting}

frameshape=\{(top shape)\}(left shape)\{(right shape)\}(bottom shape)
gives you full control over the drawn frame parts. The arguments are not case sensitive.

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

(top shape) and (bottom shape) are ‘left-rule-right’ sequences (or empty). The first ‘left-rule-right’ sequence is attached to the most inner rule, the second to the next, and so on. Each sequence has three characters: ‘rule’ is either y or n; ‘left’ and ‘right’ are y, n or r (which makes a corner round). The example uses RYRNYYYY 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=(RYRNYYYY){yny}{yny}{RYRNYYYY}}

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

4.3.11 Indexing

index=\[(\langle number\rangle)\](\langle keyword classes\rangle)\{(identifiers)\}
moreindex=\[(\langle number\rangle)\](\langle keyword classes\rangle)\{(identifiers)\}
deleteindex=\[(\langle number\rangle)\](\langle keyword classes\rangle)\{(identifiers)\}

define, add and remove (identifiers) and (keyword classes) from the index class list \langle number\rangle. If you don’t specify the optional number, the package assumes \langle number\rangle = 1.

Each appearance of the explicitly given identifiers and each appearance of the identifiers of the specified \langle keyword classes\rangle is indexed. For example, you could write index=\{1\}[keywords] to index all keywords. Note that \{1\} is required here—otherwise we couldn’t use the second optional argument.

indexstyle=\{(\langle number\rangle)\}\{\langle tokens (one-parameter command)\}\ \lstindexmacro
\langle tokens\rangle actually indexes the identifiers for the list \langle number\rangle. In contrast to the style keys, \langle tokens\rangle must read exactly one parameter, namely the identifier. Default definition is\lstindexmacro

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

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

Section 2.9 describes this feature in detail.
4.3.12 Column alignment

(columns=[(c|l|r)](alignment))

selects the column alignment. The (alignment) can be fixed, flexible, spaceflexible, or fullflexible; see section 2.10 for details.

The optional c, l, or r controls the horizontal orientation of smallest output units (keywords, identifiers, etc.). The arguments work as follows, where vertical bars visualize the effect: |listing|, |listing|, and |listing| in fixed column mode, |listing|, |listing|, and |listing| with flexible columns, and |listing|, |listing|, and |listing| with space-flexible or full flexible columns (which ignore the optional argument, since they do not add extra space around printable characters).

(flexiblecolumns=(true|false)) or flexiblecolumns=false

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

\lstset{commentstyle=\scriptsize}

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

\end{document}
{ scriptsize font
looks better now }

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

4.3.13 Escaping to \LaTeX

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

\texttt{texcl=\{true|false\}} or \texttt{texcl=false}
activates or deactivates \LaTeX comment lines. If activated, comment line delimiters are printed as usual, but the comment line text (up to the end of line) is read as \LaTeX code and typeset in comment style.

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

// calculate $a_{ij}$
\end{lstlisting}

\texttt{mathescape=\{true|false\}} or \texttt{mathescape=false}
activates or deactivates special behaviour of the dollar sign. If activated a dollar sign acts as \TeX’s text math shift.

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

\texttt{escapechar=\{character\}} or \texttt{escapechar=\{\}}

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

\texttt{escapeinside=\{character\}\{character\}} or \texttt{escapeinside=\{\}}

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

\texttt{escapebegin=\{tokens\}}

\texttt{escapeend=\{tokens\}}

The tokens are executed at the beginning respectively at the end of each escape, in particular for \texttt{texcl}. See section 8 for an application.
In the first example the comment line up to $a_{ij}$ has been typeset by the listings package in comment style. The $a_{ij}$ itself is typeset in ‘\TeX math mode’ without comment style. About half of the comment line of the second example has been typeset by this package, and the rest is in ‘\LaTeX mode’.

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

1. Don’t use any commands 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, unless they are \global.

4. \if \else \fi, groups, math shifts $\$ and $$\ldots must be balanced within each escape.

5. . . .

Expand that list yourself and mail me about new items.

4.4 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 will implement them ;-) .

\texttt{fancyvrb=(true|false)}

activates or deactivates the interface. If active, verbatim code is read by fancyvrb but typeset by listings, i.e. with emphasized keywords, strings, comments, and so on. Internally we use a very special definition of \FancyVerbFormatLine.

This interface works with Verbatim, BVerbatim and LVerbatim. But you shouldn’t use fancyvrb’s defineactive. (As far as I can see it doesn’t matter since it does nothing at all, but for safety . . . .) If fancyvrb and listings provide similar functionality, you should use fancyvrb’s.

\texttt{fvcmdparams=(command1)(number1)\ldots}

\texttt{\overlay1}
If you use `fancyvrb`'s `commandchars`, you must tell the `listings` package how many arguments each command takes. If a command takes no arguments, there is nothing to do.

The first (third, fifth, ...) parameter to the keys is the command and the second (fourth, sixth, ...) is the number of arguments that command takes. So, if you want to use \textcolor{red}{keyword} with the `fancyvrb`-`listings` interface, you should write \lstset{morefvcmdparams=\textcolor{2}}.

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

### 4.5 Environments

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

\lstnewenvironment
\begin{pascal}
\end{pascal}

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

\lstnewenvironment{pascal}
\begin{pascal}
\end{pascal}

\begin{Verbatim}
\begin{pascal}
\end{pascal}
\end{Verbatim}
Doing other things is as easy, for example, using more keys and adding an optional argument to adjust settings each listing:

```latex
\lstnewenvironment{pascalx}[1][]
  \{\lstset{language=pascal,numbers=left,numberstyle=\tiny,fontfamily=\ttfamily,fontseries=\itshape,fontshape=\sf,fontsize=\scriptsize,#1}}
  \}
```

4.6 Short Inline Listing Commands

Short equivalents of `\lstinline` can also be defined, in a manner similar to the short verbatim macros provided by `shortvrb`.

```latex
\lstMakeShortInline[[\textit{options}]]\textbackslash{character}
```

defines \texttt{character} to be an equivalent of `\lstinline[[\textit{options}]]\textbackslash{character}`, allowing for a convenient syntax when using lots of inline listings.

```latex
\lstDeleteShortInline\textbackslash{character}
```

removes a definition of \texttt{character} created by `\lstMakeShortInline`, and returns \texttt{character} to its previous meaning.

4.7 Language definitions

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

```
\lstdefinelanguage[\textit{dialect}]{\textit{language}}
{\textit{base dialect}}{\textit{and base language}}
\textit{key=value list}
{\textit{list of required aspects (keyword,comments,etc.)}}
```

defines the (given dialect of the) programming language \texttt{language}. If the language definition is based on another definition, you must specify the whole `\lstdefinelanguage` command. Note that an empty `\textit{base dialect}` uses the default dialect.

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

```
\lst@definelanguage
```

has the same syntax and is used to define languages in the driver files.

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

```
\lstalias\{\textit{alias}\}\{\textit{language}\}
```

defines an alias for a programming language. Each \texttt{alias} is redirected to the same dialect of \texttt{language}. It’s also possible to define an alias for one particular dialect only.
Here all four parameters are nonoptional and an alias with empty \{dialect\}
will select the default dialect. Note that aliases cannot be chained: The
two aliases \lstalias{foo1}{foo2} and \lstalias{foo2}{foo3} will
not redirect \texttt{foo1} to \texttt{foo3}.

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

**Keywords** We begin with keyword building keys. Note: *If you want to enter*
\, \{, \}, \%, \# or & as (part of) an argument to the keywords below, you must do it
with a preceding backslash!

\begin{verbatim}
\lstalias{\{\alias \{\dialect \} \{\language \}}
\end{verbatim}

\texttt{keywords}=[\langle \text{number} \rangle \{\langle \text{list of keywords} \rangle \}]

\texttt{morekeywords}=[\langle \text{number} \rangle \{\langle \text{list of keywords} \rangle \}]

\texttt{deletekeywords}=[\langle \text{number} \rangle \{\langle \text{list of keywords} \rangle \}]

define, add to or remove the keywords from keyword list \langle \text{number} \rangle. The use
of \texttt{keywords} is discouraged since it deletes all previously defined keywords
in the list and is thus incompatible with the \texttt{alsolanguage} key.

Please note the keys \texttt{alsoletter} and \texttt{alsodigit} below if you use unusual
charaters in keywords.

\begin{verbatim}
\deprecated ndkeywords=\{\langle \text{list of keywords} \rangle \} \texttt{ndkeywords}
\end{verbatim}

\begin{verbatim}
\deprecated morendkeywords=\{\langle \text{list of keywords} \rangle \} \texttt{morendkeywords}
\end{verbatim}

\begin{verbatim}
\deprecated deletendkeywords=\{\langle \text{list of keywords} \rangle \} \texttt{deletendkeywords}
\end{verbatim}

define, add to or remove the keywords from keyword list 2; note that this
is equivalent to \texttt{keywords}[2]...etc. The use of \texttt{ndkeywords} is strongly
discouraged.

\begin{verbatim}
\addontextcs=[\langle \text{class number} \rangle \{\langle \text{list of control sequences (without backslashes)} \rangle \} \texttt{addontextcs}
\end{verbatim}

\begin{verbatim}
\addontextcs=[\langle \text{class number} \rangle \{\langle \text{list of control sequences (without backslashes)} \rangle \} \texttt{addontextcs}
\end{verbatim}

Ditto for control sequences in \LaTeX and \input{\LaTeX}.

\begin{verbatim}
\optional directives=\{\langle \text{list of compiler directives} \rangle \} \texttt{optional directives}
\end{verbatim}

\begin{verbatim}
\optional moredirectives=\{\langle \text{list of compiler directives} \rangle \} \texttt{optional moredirectives}
\end{verbatim}
Table 2: Standard character table

<table>
<thead>
<tr>
<th>class</th>
<th>characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>A B C D E F G H I J K L M N O P Q R S T U V W X Y Z</td>
</tr>
<tr>
<td></td>
<td>a b c d e f g h i j k l m n o p q r s t u v w x y z</td>
</tr>
<tr>
<td>@</td>
<td>$</td>
</tr>
<tr>
<td>_</td>
<td></td>
</tr>
</tbody>
</table>

| digit | 0 1 2 3 4 5 6 7 8 9 |
| other | ! " # % & ' ( ) * + , - . / ; < = ? |
| space | chr(32) |
| tabulator | chr(9) |
| form feed | chr(12) |

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

optional deletedirectives={⟨list of compiler directives⟩}

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

sensitive={true|false}

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

alsoletter={⟨character sequence⟩}

alsodigit={⟨character sequence⟩}

alsoother={⟨character sequence⟩}

All identifiers (keywords, directives, and such) consist of a letter followed by alpha-numeric characters (letters and digits). For example, if you write keywords={one-two, \#include}, the minus sign must become a digit and the sharp a letter since the keywords can’t be detected otherwise.

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

otherkeywords={⟨keywords⟩}

Defines keywords that contain other characters, or start with digits. Each given ‘keyword’ is printed in keyword style, but without changing the ‘letter’, ‘digit’ and ‘other’ status of the characters. This key is designed to define keywords like =>, ->, -->, <<- and so on. If one keyword is a subsequence of another (like -- and --), you must specify the shorter first.

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

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

\texttt{string=}\begin{cases} (\text{delimiter} (\text{character})) \end{cases}
\texttt{morestring=}\begin{cases} (\text{delimiter}) \end{cases}
\texttt{deletestring=}\begin{cases} (\text{delimiter}) \end{cases}

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

The optional argument is the type and controls the how the delimiter itself is represented in a string or character literal: it is escaped by a backslash, doubled (or both is allowed via bd). Alternately, the type can refer to an unusual form of delimiter: string delimiters (akin to the s comment type) or matlab-style delimiters. The latter is a special type for Ada and Matlab and possibly other languages where the string delimiters are also used for other purposes. It is equivalent to d, except that a string does not start after a letter, a right parenthesis, a right bracket, or some other characters.

Comments

\texttt{comment=}\begin{cases} (\text{type}) (\text{delimiter}(s)) \end{cases}
\texttt{morecomment=}\begin{cases} (\text{type})(\text{delimiter}(s)) \end{cases}
\texttt{deletecomment=}\begin{cases} (\text{type})(\text{delimiter}(s)) \end{cases}

\begin{itemize}
\item Ditto for comments, but some types require more than a single delimiter.
\item The following overview uses \texttt{morecomment} as the example, but the examples apply to \texttt{comment} and \texttt{deletecomment} as well.
\end{itemize}

\texttt{morecomment=}\begin{cases} [l] (\text{delimiter}) \end{cases}

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

\texttt{morecomment=}\begin{cases} [s] \{ (\text{delimiter}) \{ (\text{delimiter}) \end{cases}

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

\texttt{morecomment=}\begin{cases} [n] \{ (\text{delimiter}) \{ (\text{delimiter}) \end{cases}

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

\texttt{morecomment=}\begin{cases} [f] \{ (\text{delimiter}) \end{cases}
morecomment=[f][commentstyle]([n=preceding columns])\{delimiter\}

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

optional keywordcomment={⟨keywords⟩}
optional morekeywordcomment={⟨keywords⟩}
optional deletekeywordcomment={⟨keywords⟩}

A keyword comment begins with a keyword and ends with the same keyword.
Consider keywordcomment={comment,co}. Then ‘comment…comment’
and ‘co…co’ are comments.

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

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

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

optional podcomment={true|false}
activates or deactivates PODs—Perl specific.

4.8 Installation

Software installation

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

listings.pdf \rightarrow texmf/doc/latex/listings
listings.dtx, listings.ins,
listings.ind, lstpatch.sty,
lstdrvrs.dtx \rightarrow texmf/source/latex/listings

Note that you may not 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, if it exists already,
remove all files except lst⟨whatever⟩0.sty and lstlocal.cfg from it.

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.

48
If your \TeX implementation uses a file name database, update it.

If you receive a patch file later on, put it where \texttt{listings.sty} is (and update the file name database).

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

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

Never modify a file from the \texttt{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

\begin{verbatim}
  lstmisc0.sty  for local add-ons (see the developer’s guide),
  lstlang0.sty  for local language definitions (see 4.7), and
  lstlocal.cfg  as local configuration file
\end{verbatim}

and put them in the same directory as the other \texttt{listings} files. These three files are not touched by a new installation unless you remove them. If \texttt{lstlocal.cfg} exists, it is loaded after \texttt{listings.cfg}. You might want to change one of the following parameters.

\begin{verbatim}
data \lstaspectfiles contains lstmisc0.sty,lstmisc.sty
data \lstlanguagefiles contains lstlang0.sty,lstlang1.sty,lstlang2.sty,lstlang3.sty
\end{verbatim}

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

Moreover, you might want to adjust \texttt{\lstlistlistingname}, \texttt{\lstlistingname}, \texttt{\lstlistlistingnamestyle}, \texttt{\defaultdialect}, \texttt{\lstalias}, or \texttt{\lstalias} as described in earlier sections.

5 **Experimental features**

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

5.1 **Listings inside arguments**

There are some things to consider if you want to use \texttt{\lstinline} or the listing environment inside arguments. Since \TeX reads the argument before the ‘\texttt{lst-}' macro’ is executed, this package can’t do anything to preserve the input: spaces
shrink to one space, the tabulator and the end of line are converted to spaces, \TeX’s comment character is not printable, and so on. Hence, you must work a bit more. You have to put a backslash in front of each of the following four characters: \%, \{, \}, and %. 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\footnote{\lstinline{var i:integer;} and \lstinline!protected\ \ spaces! and \fbox{\lstinline!\\{%}}} if the current language is Pascal. Note that this example shows another experimental feature: use of argument braces as delimiters. This is described in section 4.2.

And now an environment example:

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

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

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, for instance, each function or procedure definition and variable declaration is preceded by a particular keyword. Note that you must request the following keys with the procnames option: \usepackage[procnames]{listings}.

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

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

\footnote{\ lstinline{var i:integer;} and protected \ \ spaces and %}
defines the style in which procedure and function names appear.

If activated, procedure and function names are also indexed.

To do: The procnames aspect is unsatisfactory (and has been unchanged at least since 2000). It marks and indexes the function definitions so far, but it would be possible to mark also the following function calls, for example. A key could control whether function names are added to a special keyword class, which then appears in 'procname' style. But should these names be added globally? There are good reasons for both. Of course, we would also need a key to reset the name list.

5.3 Hyperlink references

This very small aspect must be requested via the hyper option since it is experimental. One possibility 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.

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

set a hyperlink anchor and link, respectively. The defaults are suited for the hyperref package.

5.4 Literate programming

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

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

Funny, isn’t it? We could leave \( i := 0 \) in our listings instead of \( i \leftarrow 0 \), 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)\ldots(replacement item)

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

Each ‘printing unit’ in \( \langle \text{replacement text} \rangle \) must be in braces unless 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\$\}{\$\rightarrow\$\}}3. Note the braces around the arrows.

If one \( \langle \text{replace} \rangle \) is a subsequence of another \( \langle \text{replace} \rangle \), you must define the shorter sequence first. For example, \{-\} must be defined before \{--\} and this before \{-\}.

The optional star indicates that literate replacements should not be made in strings, comments, and other delimited text.

In the example above, I’ve used

\[
\%
\text{literate}={:=}\{\{\$\text{gets}\}\}1 \{<=\}\{\{\$\text{leq}\}\}1 \{>=\}\{\{\$\text{geq}\}\}1 \{<>\}\{\{\$\text{neq}\}\}1
\]

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

### 5.5 LGrind definitions

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

\textit{optional lgrindef=⟨language⟩}

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

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

\textit{data,optional \texttt{lstatg}lgrindef\texttt{file}}

\texttt{lgrindef}. contains the (path and) name of the definition file.

### 5.6 † Automatic formatting

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

\[
\langle \text{input chars} \rangle = \langle \text{exceptional chars} \rangle \langle \text{pre} \rangle \langle \text{string} \rangle \langle \text{post} \rangle
\]
Whenever (input chars) aren’t followed by one of the (exceptional chars), formatting is done according to the rest of the value. If \texttt{string} isn’t specified, the input characters aren’t printed (except it’s an identifier or keyword). Otherwise \texttt{(pre)} is ‘executed’ before printing the original character string and \texttt{(post)} afterwards. These two are ‘subsets’ of

- \texttt{\newline} — ensuring a new line;
- \texttt{\space} — ensuring a whitespace;
- \texttt{\indent} — increasing indentation;
- \texttt{\noindent} — decreasing indentation.

Now we can give an example.

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

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

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

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

1. Insert horizontal space to separate function/procedure name and following parenthesis or to separate arguments of a function, e.g. add the space after a comma (if inside function call).

2. Smart breaking of long lines. Consider long ‘and/or’ expressions. Formatting should follow the logical structure!

3. Context sensitive formatting rules. It can be annoying if empty or small blocks take three or more lines in the output—think of scrolling down all the time. So it would be nice if the block formatting was context sensitive.
Note that this is a very first and clumsy attempt to provide automatic formatting—clumsy since the problem isn’t trivial. Any ideas are welcome. Implementations also. Eventually you should know that you must request format definitions at package loading, e.g. via `\usepackage[formats]{listings}.

### 5.7 Arbitrary linerrange markers

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

```latex
\lstset{rangeprefix=\{,% curly left brace plus space
 rangesuffix=\ \}}% space plus curly right brace
{ loop 2 }
for i:=maxint to 0 do
begin
  { do nothing }
end;
{ end }
{ loop 2 }
for i:=maxint to 0 do
begin
  { do nothing }
end;
{ end }
\end{lstlisting}
```

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

- `rangeprefix=⟨prefix⟩`
- `rangesuffix=⟨suffix⟩`
- `rangefileprefix=⟨prefix⟩`
- `rangeendsuffix=⟨suffix⟩`

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

- `rangeprefix=⟨prefix⟩`
- `rangesuffix=⟨suffix⟩`

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

- `includeangemarker=⟨true|false⟩`  
  shows or hides the markers in the output.
for \texttt{i:=maxint to 0} do
begin
\{ \texttt{do nothing} \}
end;

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

5.8 Multicolumn Listings

When the \texttt{multicol} package is loaded, it can be used to typeset multi-column listings. These are specified with the \texttt{multicols} key. For example:

\begin{lstlisting}[
\texttt{multicols=2}]
if (i < 0)
\texttt{i = 0}
\texttt{j = 1}
end if
\end{lstlisting}

The multicolumn option is known to fail with some keys.

→ Which keys? Unfortunately, I don’t know. Carsten left the code for this option in the version 1.3b patch file with only that cryptic note for documentation. Bug reports would be welcome, though I don’t promise that they’re fixable. —Brooks

Tips and tricks

Note: This part of the documentation is under construction. Section 8 must be sorted by topic and ordered in some way. Moreover a new section ‘Examples’ is planned, but not written. Lack of time is the main problem ...

6 Troubleshooting

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

1. Start from the minimal file in section 1.1.

2. Add the \LaTeX code which causes the problem, but keep it short. In particular, keep the number of additional packages small.

3. Remove some code from the file (and the according packages) until the problem disappears. Then you’ve found a crucial piece.

55
4. Add this piece of code again and start over with step 3 until all code and all packages are substantial.

5. You now have a minimal file. Send a bug report to the address on the first page of this documentation and include the minimal file together with the created `.log`-file. If you use a very special package (i.e. one not on CTAN), also include the package if its software license allows it.

7 Bugs and workarounds

7.1 Listings inside arguments

At the moment it isn’t possible to use `\lstinline{...}` in a cell of a table (see section 18.4.1 on page 196 for more information), but it is possible to define a wrapper macro which can be used instead of `\lstinline{...}`:

```latex
\newcommand\foo{\lstinline{t}}
\newcommand\foobar[2][]{\lstinline{[#1]{#2}}}
```

| \begin{tabular}{ll}
| foo & a variable \\
| \foobar[language=java]{int u;} & a declaration
| \end{tabular} |
|---|---|
| t | a variable |
| int u; | a declaration |

7.2 Listings with a background colour and \LaTeX escaped formulas

If there is any text escaped to \LaTeX with some coloured background and surrounding frames, then there are gaps in the background as well as in the lines making up the frame.

```latex
\begin{lstlisting}[language=C, mathescape, backgroundcolor=\color{yellow!10}, frame=tlb]
/* the following code computes $\sum_{i=1}^{n}i$ */
for (i = 1; i <= limit; i++) {
    sum += i;
}
\end{lstlisting}
```

At the moment there is only one workaround:

- Write your code into an external file *(filename)*.
• Input your code by `\lstinputlisting{filename}` into your document and surround it with a frame generated by `\begin{mdframed} ... \end{mdframed}.

```
\begin{verbatimwrite}{temp.c}
/* the following code computes \[ \sum_{i=1}^{n}i \] */
for (i = 1; i <= limit; i++) {
    sum += i;
}
\end{verbatimwrite}
\begin{mdframed}[backgroundcolor=yellow!10, rightline=false]
\lstinputlisting[language=C,mathescape,frame={}]{./temp.c}
\end{mdframed}
```

/* the following code computes \[ \sum_{i=1}^{n}i \] */
for (i = 1; i <= limit; i++) {
    sum += i;
}

For more information about the `verbatimwrite` environment have a look at [Fai11], the `mdframed` environment is deeply discussed in [DS13].

8 How tos

How to reference line numbers

Perhaps you want to put `\label{⟨whatever⟩}` into a LaTeX escape which is inside a comment whose delimiters aren’t printed? If you did that, the compiler won’t see the LaTeX code since it would be inside a comment, and the listings package wouldn’t print anything since the delimiters would be dropped and `\label` doesn’t produce any printable output, but you could still reference the line number. 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 }
end;
\end{lstlisting}
```

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=oftware
Here \"M represents the end of line character.

How to gobble characters

To make your \LaTeX code more readable, you might want to indent your \lstlisting listings. This indention should not show up in the pretty-printed listings, however, so it must be removed. If you indent each code line by three characters, you can remove them via \begin{lstlisting}[gobble=3]

\begin{lstlisting}[gobble=3]
for \i:=\text{maxint} to 0 do
begin
{ do nothing }
end;
\end{lstlisting}

Note that empty lines and the beginning and the end of the environment need not respect the indention. However, never indent the end by more than \begin{lstlisting}[gobble=3] characters. Moreover note that tabulators expand to tabsize spaces before we gobble.

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

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

How to include graphics

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

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

The following example is originally from a header file I got from Herbert. For the presentation here I use the \lstlisting environment and an excerpt from the header file. The \begin{lstlisting}[mathescape=true]
% /* $ \end{lstlisting}
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;

How to get closed frames on each page

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

```latex
\begin{framed}
\begin{lstlisting}
\end{lstlisting}
\end{framed}
```

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

How to print national characters with Λ and listings

Apart from typing in national characters directly, you can use the ‘escape’ feature described in section 4.3.13. The keys `escapechar`, `escapeinside`, and `texcl` allow partial usage of \LaTeX code.

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

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

```latex
\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`:

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

```latex
\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 pro-
gramming language are single characters, like the braces in Pascal:

```
\lstset{escapebegin=\textbraceleft\begin{arab}, escapeend=\end{arab}\textbraceright}
\begin{lstlisting}[escapeinside=\{}\]
for i:=maxint to 0 do begin
  \{ Replace text by Arabic comment. \}
end;
\end{lstlisting}
```

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

**How to get bold typewriter type keywords**

Use the **LuxiMono** package.

**How to work with plain text**

If you want to use listings to set plain text (perhaps with line numbers, or like `verbatim` but with line wrapping, or so forth, use the empty language:

```
\lstset{language=}
```

**How to get the developer’s guide**

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

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

Then run `listings.dtx` through `\LaTeX` twice, run Makeindex (with the `-s gind.ist` option), and then run `\LaTeX` one last time on `listings.dtx`. This creates the whole documentation including User’s guide, Reference guide, Developer’s guide, and Implementation.

If you can run the (GNU) make program, executing the command

```
make all
```
or

```
make listings-devel.pdf
```
or

```
make pdf-devel
```
gives the same result—it is called `listings-devel.pdf`.

**Developer’s guide**

First I must apologize for this developer’s guide since some parts are not explained as well as possible. But note that you are in a pretty good shape: this developer’s guide exists! You might want to peek into section 10 before reading section 9.
9 Basic concepts

The functionality of the listings package appears to be divided into two parts: on the one hand commands which actually typeset listings and on the other via \lstset adjustable parameters. Both could be implemented in terms of lst-aspects, which are simply collections of public keys and commands and internal hooks and definitions. The package defines a couple of aspects, in particular the kernel, the main engine. Other aspects drive this engine, and language and style definitions tell the aspects how to drive. The relations between car, driver and assistant driver are exactly reproduced—and I’ll be your driving instructor.

9.1 Package loading

Each option in \usepackage[⟨options⟩]{listings} loads an aspect or prevents the package from loading it if the aspect name is preceded by an exclamation mark. This mechanism was designed to clear up the dependencies of different package parts and to debug the package. For this reason there is another option:

\begin{verbatim}
\lstloadaspects{⟨comma separated list of aspect names⟩}
\end{verbatim}

loads the specified aspects if they are not already loaded.

Here now is a list of all aspects and related keys and commands—in the hope that this list is complete.

strings

\begin{verbatim}
string, morestring, deletestring, stringstyle, showstringspaces
\end{verbatim}

comments

\begin{verbatim}
comment, morecomment, deletecomment, commentstyle
\end{verbatim}

pod

\begin{verbatim}
printpod, podcomment
\end{verbatim}

escape

\begin{verbatim}
texcl, escapebegin, escapeend, escapechar, escapeinside, mathescape
\end{verbatim}

writefile requires 1 \toks, 1 \write

\begin{verbatim}
\lst@BeginWriteFile, \lst@BeginAlsoWriteFile, \lst@EndWriteFile
\end{verbatim}

style

\begin{verbatim}
empty style, style, \lstdefinestyle, \lst@definestyle, \lst@definestylefiles
\end{verbatim}
language
  empty language, language, alsolanguage, defaultedialect, \lstalias, \lstdefinelanguage, \lst@definelanguage, \lstloadlanguages, \lstlanguagefiles

keywords
  sensitive, classoffset, keywords, morekeywords, deletekeywords, keywordstyle, ndkeywords, morendkeywords, deletendkeywords, ndkeywordstyle, keywordsprefix, otherkeywords

emph requires keywords
  emph, moreemph, deleteemph, emphstyle

html requires keywords
  tag, usekeywordsintag, tagstyle, markfirstintag

tex requires keywords
  texcs, moretexcs, deletetexcs, texcsstyle

directives requires keywords
  directives, moredirectives, deletedirectives, directivestyle

index requires keywords
  index, moreindex, deleteindex, indexstyle, \lstindexmacro

procnames requires keywords
  procnamestyle, indexprocnames, procnamekeys, moreprocnamekeys, deleteprocnamekeys

keywordcomments requires keywords, comments
  keywordcomment, morekeywordcomment, deletekeywordcomment, keywordcommentssemicolon

labels requires 2 \count
  numbers, numberstyle, numbersep, stepnumber, numberblanklines, firstnumber, \thelstnumber, numberfirstline

lineshape requires 2 \dimen
  xleftmargin, xrightmargin, resetmargins, linewidth, lineskip, breaklines, breakindent, breakautoindent, prebreak, postbreak, breakatwhitespace

frames requires lineshape
  framexleftmargin, framexrightmargin, frametopmargin, framexbottommargin, backgroundcolor, fillcolor, rulecolor, rulesepcolor, rulesep, framerate, framesep, frameshape, frameround, frame

make requires keywords
  makemacrouse
The kernel allocates 6 \texttt{count}, 4 \texttt{dimen} and 1 \texttt{toks}. Moreover it defines the following keys, commands, and environments:

\begin{itemize}
\item basewidth, \texttt{fontadjust}, \texttt{columns}, \texttt{flexiblecolumns}, \texttt{identifierstyle}, \texttt{tabsize}, \texttt{showtabs}, \texttt{tab}, \texttt{showspaces}, \texttt{keepspace}, \texttt{formfeed}, \texttt{SelectCharTable}, \texttt{MoreSelectCharTable}, \texttt{extendedchars}, \texttt{alsoletter}, \texttt{alsodigit}, \texttt{alsoother}, \texttt{excludedelims}, \texttt{literate}, \texttt{basicstyle}, \texttt{print}, \texttt{firstline}, \texttt{lastline}, \texttt{linerange}, \texttt{nolol}, \texttt{captionpos}, \texttt{abovecaptionskip}, \texttt{belowcaptionskip}, \texttt{label}, \texttt{title}, \texttt{caption}, \texttt{\lstlistingname}, \texttt{\lstlistingnamestyle}, \texttt{boxpos}, \texttt{float}, \texttt{floatplacement}, \texttt{aboveskip}, \texttt{belowskip}, \texttt{everydisplay}, \texttt{showlines}, \texttt{emptylines}, \texttt{gobble}, \texttt{name}, \texttt{\lstname}, \texttt{\lstlistoflistings}, \texttt{\lstnewenvironment}, \texttt{\linenl}, \texttt{\lstinputlisting}, \texttt{\lstlisting}, \texttt{\lstloadaspects}, \texttt{\lstset}, \texttt{\lstaspectfiles}, \texttt{inputencoding}, \texttt{inputpath}, \texttt{delim}, \texttt{moredelim}, \texttt{deletedelim}, \texttt{upquote}, \texttt{numberbychapter}, \texttt{\lstMakeShortInline}, \texttt{\lstDeleteShortInline}, \texttt{fancyvrb}
\end{itemize}

\section{How to define \texttt{lst}-aspects}

There are at least three ways to add new functionality: (a) you write an aspect of general interest, send it to me, and I’ll just paste it into the implementation; (b) you write a ‘local’ aspect not of general interest; or (c) you have an idea for an aspect and make me writing it. (a) and (b) are good choices.

An aspect definition starts with \texttt{\lst@BeginAspect} plus arguments and ends with the next \texttt{\lst@EndAspect}. In particular, aspect definitions can’t be nested.

\begin{verbatim}
\lst@BeginAspect[[\texttt{list of required aspects}]]\{\texttt{aspect name}\}
\lst@EndAspect
\end{verbatim}

The optional list is a comma separated list of required aspect names. The complete aspect is not defined in each of the following cases:

1. \texttt{(aspect name)} is empty.
2. The aspect is already defined.
3. A required aspect is neither defined nor loadable via \texttt{\lstloadaspects}.
Consequently you can’t define a part of an aspect and later on another part. But it is possible to define aspect $A_1$ and later aspect $A_2$ which requires $A_1$.

→ Put local add-ons into ‘lstmisc0.sty’—this file is searched first by default. If you want to make add-ons for one particular document just replace the surrounding ‘\lst@BeginAspect’ and ‘\lst@EndAspect’ by ‘makeatletter’ and ‘makeatother’ and use the definitions in the preamble of your document. However, you have to load required aspects on your own.

You can put any TeX material in between the two commands, but note that definitions must be ‘global’ if you need them later—LATEX’s ‘newcommand’ makes local definitions and can’t be preceded by ‘global’. So use the following commands, ‘\gdef’, and commands described in later sections.

\lst@UserCommand{〈macro〉}{⟨parameter text⟩}{⟨replacement text⟩}

The macro is (mainly) equivalent to ‘\gdef’. The purpose is to distinguish user commands and internal global definitions.

\lst@Key{⟨key name⟩}{⟨init value⟩}[[⟨default value⟩]]{⟨definition⟩}

\lst@Key{⟨key name⟩}{relax}[[⟨default value⟩]]{⟨definition⟩}

defines a key using the keyval package from David Carlisle. 〈definition〉 is the replacement text of a macro with one parameter. The argument is either the value from ‘key=value’ or ‘default value’ if no ‘=value’ is given. The helper macros \lstKV@... below might simplify 〈definition〉.

The key is not initialized if the second argument is ‘relax’. Otherwise ‘init value’ is the initial value given to the key. Note that we locally switch to ‘\globalsdefs=1’ to ensure that initialization is not effected by grouping.

\lst@AddToHook{〈name of hook〉}{⟨TeX material⟩}

adds TeX material at predefined points. Section 9.4 lists all hooks and where they are defined respectively executed. \lst@AddToHook{A}{\csa} before \lst@AddToHook{A}{\csb} does not guarantee that \csa is executed before \csb.

\lst@AddToHookExe{〈name of hook〉}{⟨TeX material⟩}

also executes ⟨TeX material⟩ for initialization. You might use local variables—local in the sense of TeX and/or usual programming languages—but when the code is executed for initialization all assignments are global: we set ‘\globalsdefs=1’ locally to one.

\lst@UseHook{〈name of hook〉}

executes the hook.

→ Let’s look at two examples. The first extends the package by adding some hook-material. If you want status messages, you might write

% \lst@AddToHook{Init}{\message{\MessageBreak Processing listing ...}}
% \lst@AddToHook{DeInit}{\message{complete.\MessageBreak}}

The second example introduces two keys to let the user control the messages. The macro \lst@BeginAspect is described in section 11.1.

% \lst@BeginAspect{message}
% \lst@Key{message}{Annoying message.}{\gdef\lst@message{#1}}
% \lst@Key{moremessage}{relax}{\lst@AddTo\lst@message{\MessageBreak#1}}
% \lst@AddToHook{Init}{\typeout{\MessageBreak\lst@message}}
% \lst@EndAspect
However, there are certainly aspects which are more useful.

The following macros can be used in the (definition) argument of the \lst@Key command to evaluate the argument. The additional prefix KV refers to the \texttt{keyval} package.

\lstKV@SetIf{\langle value\rangle}{\langle if macro\rangle}

\texttt{(if macro)} becomes \texttt{\textbackslash iftrue} if the first character of \texttt{\langle value\rangle} equals \texttt{t} or \texttt{T}. Otherwise it becomes \texttt{\textbackslash iffalse}. Usually you will use \texttt{#1} as \texttt{\langle value\rangle}.

\lstKV@SwitchCases{\langle value\rangle}

\texttt{(string 1)}&\texttt{(execute 1)}\\
\texttt{(string 2)}&\texttt{(execute 2)}\\
\vdots\\
\texttt{(string n)}&\texttt{(execute n)}{\langle else\rangle}

Either execute \texttt{\langle else\rangle} or the \texttt{\langle value\rangle} matching part.

This implementation of C. Heinz has a problem, if the listing is part of a tabular environment as found out by Nasser M. Abbasi. David Carlisle gave a hint how to avoid this problem and so the separator & is replaced by :.

\lstKV@TwoArg{\langle value\rangle}{\langle subdefinition\rangle}

\lstKV@ThreeArg{\langle value\rangle}{\langle subdefinition\rangle}

\lstKV@FourArg{\langle value\rangle}{\langle subdefinition\rangle}

\texttt{\langle subdefinition\rangle} is the replacement text of a macro with two, three, and four parameters. We call this macro with the arguments given by \texttt{\langle value\rangle}. Empty arguments are added if necessary.

\lstKV@OptArg{\langle default arg.\rangle}{\langle value\rangle}{\langle subdefinition\rangle}

\texttt{\langle default arg.\rangle} is not optional. \texttt{\langle subdefinition\rangle} is the replacement text of a macro with parameter text \texttt{##1##2}. Note that the macro parameter character # is doubled since used within another macro. \texttt{\langle subdefinition\rangle} accesses these arguments via \texttt{##1} and \texttt{##2}.

\texttt{\langle value\rangle} is usually the argument \texttt{#1} passed by the \texttt{keyval} package. If \texttt{\langle value\rangle} has no optional argument, \texttt{\langle default arg.\rangle} is inserted to provide the arguments to \texttt{\langle subdefinition\rangle}.

\lstKV@XOptArg{\langle default arg.\rangle}{\langle value\rangle}{\langle submacro\rangle}

Same as \lstKV@OptArg but the third argument \texttt{\langle submacro\rangle} is already a definition and not replacement text.

\lstKV@CSTwoArg{\langle value\rangle}{\langle subdefinition\rangle}

\texttt{\langle value\rangle} is a comma separated list of one or two arguments. These are given to the subdefinition which is the replacement text of a macro with two parameters. An empty second argument is added if necessary.

→ One more example. The key 'sensitive' belongs to the aspect keywords. Therefore it is defined in between '\lst@BeginAspect{keywords}' and '\lst@EndAspect', which is not shown here.
The last line is equivalent to

\lst@AddToHook\{SetLanguage\}\{let\lst@ifsensitive\iftrue\}
\global\let\lst@ifsensitive\iftrue

We initialize the variable globally since the user might request an aspect in a group. Afterwards
the variable is used locally—there is no \global \langle \TeX material \rangle. Note that we could define
and init the key as follows:

\lst@Key{sensitive}t\{\lstKV@SetIf{#1}\lst@ifsensitive\}
\lst@AddToHook\{SetLanguage\}\{let\lst@ifsensitive\iftrue\}

9.3 Internal modes

You probably know \TeX’s conditional commands \ifhmode, \ifvmode, \ifmmode, and \ifinner. They tell you whether \TeX is in (restricted) horizontal or (internal)
vertical or in (nondisplay) mathematical mode. For example, true \ifhmode and
true \ifinner indicate restricted horizontal mode, which means that you are in a \hbox. The typical user doesn’t care about such modes; \TeX/\LaTeX \TeX manages
all this. But since you’re reading the developer’s guide, we discuss the analogue
for the listings package now. It uses modes to distinguish comments from strings,
‘comment lines’ from ‘single comments’, and so on.

The package is in ‘no mode’ before reading the source code. In the phase of
initialization it goes to ‘processing mode’. Afterwards the mode depends on the
actual source code. For example, consider the line

\texttt{"string" // comment}

and assume language=C++. Reading the string delimiter, the package enters
‘string mode’ and processes the string. The matching closing delimiter leaves
the mode, i.e. switches back to the general ‘processing mode’. Coming to the
two slashes, the package detects a comment line; it therefore enters ‘comment line
mode’ and outputs the slashes. Usually this mode lasts to the end of line.

But with textcl=true the escape aspect immediately leaves ‘comment line
mode’, interrupts the current mode sequence, and enters ‘\TeX comment line mode’.
At the end of line we reenter the previous mode sequence ‘no mode’ \rightarrow ‘processing
mode’. This escape to \LaTeX works since ‘no mode’ implies that \TeX’s characters
and catcodes are present, whereas ‘processing mode’ means that listings’ characters
and catcodes are active.

Table 3 lists all static modes and which aspects they belong to. Most features
use dynamically created mode numbers, for example all strings and comments.
Each aspect may define its own mode(s) simply by allocating it/them inside the
aspect definition.

\lst@NewMode{\langle mode \rangle} defines a new static mode, which is a nonnegative integer assigned to \langle mode \rangle.
\langle mode \rangle should have the prefix \lst@ and suffix mode.

\lst@UseDynamicMode{\langle token(s) \rangle} inserts a dynamic mode number as argument to the token(s).

This macro cannot be used to get a mode number when an aspect is loaded
or defined. It can only be used every listing in the process of initialization,
e.g. to define comments when the character table is selected.
Table 3: Internal modes

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Mode name</th>
<th>Usage/Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>\texttt{@nomode}</td>
<td>If this mode is active, $\LaTeX$’s ‘character table’ is present; the other implication is not true. Any other mode \textit{may} imply that catcodes and/or definitions of characters are changed.</td>
</tr>
<tr>
<td></td>
<td>\texttt{@Pmode}</td>
<td>is a general processing mode. If active we are processing a listing, but haven’t entered a more special mode.</td>
</tr>
<tr>
<td></td>
<td>\texttt{@GPmode}</td>
<td>general purpose mode for language definitions.</td>
</tr>
<tr>
<td>Pod</td>
<td>\texttt{@PODmode}</td>
<td>\ldots a POD—Perl specific.</td>
</tr>
<tr>
<td>Escape</td>
<td>\texttt{@TeXmode}</td>
<td>\ldots a comment line, but $\LaTeX$’s character table is present—except the EOL character, which is needed to terminate this mode.</td>
</tr>
<tr>
<td></td>
<td>\texttt{@TeXmode}</td>
<td>indicates that $\LaTeX$’s character table is present (except one user specified character, which is needed to terminate this mode).</td>
</tr>
<tr>
<td>Directives</td>
<td>\texttt{@CDmode}</td>
<td>indicates that the current line began with a compiler directive.</td>
</tr>
<tr>
<td>Keyword Comments</td>
<td>\texttt{@KCmode}</td>
<td>\ldots a keyword comment.</td>
</tr>
<tr>
<td></td>
<td>\texttt{@KCSmode}</td>
<td>\ldots a keyword comment which can be terminated by a semicolon only.</td>
</tr>
<tr>
<td>HTML</td>
<td>\texttt{@insidemode}</td>
<td>Active if we are between \texttt{&lt;} and \texttt{&gt;}.</td>
</tr>
<tr>
<td>Make</td>
<td>\texttt{@makemode}</td>
<td>Used to indicate a keyword.</td>
</tr>
</tbody>
</table>
changed \lst@EnterMode\{\langle start tokens\rangle\}

opens a group level, enters the mode, and executes \langle start tokens\rangle.
Use \lst@modetrue in \langle start tokens\rangle to prohibit future mode changes—except leaving the mode, of course. You must test yourself whether you’re allowed to enter, see below.

\lst@LeaveMode

returns to the previous mode by closing a group level if and only if the current mode isn’t \lst@nomode already. You must test yourself whether you’re allowed to leave a mode, see below.

\lst@InterruptModes
\lst@ReenterModes

The first command returns to \lst@nomode, but saves the current mode sequence on a special stack. Afterwards the second macro returns to the previous mode. In between these commands you may enter any mode you want. In particular you can interrupt modes, enter some modes, and say ‘interrupt modes’ again. Then two re-enters will take you back in front of the first ‘interrupt modes’.

Remember that \lst@nomode implies that \TeX’s character table is active.

Some variables show the internal state of processing. You are allowed to read them, but \texttt{direct write access is prohibited}. Note: \lst@ifmode is not obsolete since there is no relation between the boolean and the current mode. It will happen that we enter a mode without setting \lst@ifmode true, and we’ll set it true without assigning any mode!

\counter \lst@mode

keeps the current mode number. Use \ifnum\lst@mode=\langle mode name\rangle to test against a mode. Don’t modify the counter directly!

\boolean \lst@ifmode

No mode change is allowed if this boolean is true—except leaving the current mode. Use \lst@modetrue to modify this variable, but do it only in \langle start tokens\rangle.

\boolean \lst@iflmode

Indicates whether the current mode ends at end of line.

\section{Hooks}

Several problems arise if you want to define an aspect. You should and/or must (a) find additional functionality \texttt{(of general interest)} and implement it, (b) create the user interface, and (c) interface with the \texttt{listings} package, \texttt{i.e. find correct hooks and insert appropriate \TeX material}. (a) is out of the scope of this developer’s guide. The commands \lstKV@\ldots in section 9.2 might help you with (b). Here now we describe all hooks of the \texttt{listings} package.

All hooks are executed inside an overall group. This group starts somewhere near the beginning and ends somewhere at the end of each listing. Don’t make any
other assumptions on grouping. So define variables globally if it’s necessary—and be alert of side effects if you don’t use your own groups.

**AfterBeginComment**

is executed after the package has entered comment mode. The starting delimiter is usually typeset when the hook is called.

**BoxUnsafe**

Contains all material to deactivate all commands and registers which are possibly unsafe inside `\hbox`. It is used whenever the package makes a box around a listing and for `fancyvrb` support.

**DeInit**

Called at the very end of a listing but before closing the box from `BoxUnsafe` or ending a float.

**DetectKeywords**

This `Output` subhook is executed if and only if mode changes are allowed, i.e. if and only if the package doesn’t process a comment, string, and so on—see section 9.3.

**DisplayStyle**

dействивает/активирует функции для displaystyle listings.

**EmptyStyle**

Executed to select the ‘empty’ style—except the user has redefined the style.

**EndGroup**

Executed whenever the package closes a group, e.g. at end of comment or string.

**EOL**

Called at each end of input line, right before `InitVarsEOL`.

**EveryLine**

Executed at the beginning of each output line, i.e. more than once for broken lines. This hook must not change the horizontal or vertical position.

**EveryPar**

Executed once for each input line when the output starts. This hook must not change the horizontal or vertical position.

**ExitVars**

Executed right before `DeInit`.

**FontAdjust**

adjusts font specific internal values (currently `\lst@width` only).

**Init**

Executed once each listing to initialize things before the character table is changed. It is called after `PreInit` and before `InitVars`. 
**InitVars**

Called to init variables each listing.

**InitVarsBOL**

initializes variables at the beginning of each input line.

**InitVarsEOL**

updates variables at the end of each input line.

**ModeTrue**

executed by the package when mode changes become illegal. Here keyword detection is switched off for comments and strings.

**OnEmptyLine**

executed before the package outputs an empty line.

**OnNewLine**

executed before the package starts one or more new lines, i.e. before saying `\par\noindent\hbox{}` (roughly speaking).

**Output**

Called before an identifier is printed. If you want a special printing style, modify \lst@thestyle.

**OutputBox**

used inside each output box. Currently it is only used to make the package work together with Lambda—hopefully.

**OutputOther**

Called before other character strings are printed. If you want a special printing style, modify \lst@thestyle.

**PostOutput**

Called after printing an identifier or any other output unit.

**PostTrackKeywords**

is a very special Init subhook to insert keyword tests and define keywords on demand. This hook is called after TrackKeywords.

**PreInit**

Called right before Init hook.

**PreSet**

Each typesetting command/environment calls this hook to initialize internals before any user supplied key is set.

**SelectCharTable**

is executed after the package has selected the standard character table. Aspects adjust the character table here and define string and comment delimiters, and such.
SetFormat
Called before internal assignments for setting a format are made. This hook
determines which parameters are reset every format selection.

SetStyle
Called before internal assignments for setting a style are made. This hook
determines which parameters are reset every style selection.

SetLanguage
Called before internal assignments for setting a language are made. This
hook determines which parameters are reset every language selection.

TextStyle
deactivates/activates features for textstyle listings.

TrackKeywords
is a very special Init subhook to insert keyword tests and define keywords
on demand. This hook is called before PostTrackKeywords.

9.5 Character tables
Now you know how a car looks like, and you can get a driving license if you take
some practice. But you will have difficulties if you want to make heavy alterations
to the car. So let’s take a closer look and come to the most difficult part: the
engine. We’ll have a look at the big picture and fill in the details step by step. For
our purpose it’s good to override \TeX’s character table. First we define a standard
character table which contains

- letters: characters identifiers are out of,
- digits: characters for identifiers or numerical constants,
- spaces: characters treated as blank spaces,
- tabulators: characters treated as tabulators,
- form feeds: characters treated as form feed characters, and
- others: all other characters.

This character table is altered depending on the current programming language.
We may define string and comment delimiters or other special characters. Table 2
on page 46 shows the standard character table. It can be modified with the keys
alsoletter, alsodigit, and alsoother.

How do these ‘classes’ work together? Let’s say that the current character
string is ‘tr’. Then letter ‘y’ simply appends the letter and we get ‘try’. The
next nonletter (and nondigit) causes the output of the characters. Then we collect
all coming nonletters until reaching a letter again. This causes the output of the
nonletters, and so on. Internally each character becomes active in the sense of\TeX and is defined to do the right thing, e.g. we say

\% \def A{lst@ProcessLetter A}
where the first ‘A’ is active and the second has letter catcode 11. The macro \lst@ProcessLetter gets one token and treats it as a letter. The following macros exist, where the last three get no explicit argument.

\lst@ProcessLetter \langle spec. token \rangle
\lst@ProcessDigit \langle spec. token \rangle
\lst@ProcessOther \langle spec. token \rangle
\lst@ProcessTabulator
\lst@ProcessSpace
\lst@ProcessFormFeed

\langle spec. token \rangle is supposed to do two things. Usually it expands to a printable version of the character. But if \lst@UM is equivalent to \@empty, \langle spec. token \rangle must expand to a character token. For example, the sharp usually expands to \#, which is defined via \chardef and is not a character token. But if \lst@UM is equivalent to \@empty, the sharp expands to the character ‘#’ (catcode 12). Note: Changes to \lst@UM must be locally. However, there should be no need to do such basic things yourself. The listings package provides advanced macros which use that feature, e.g. \lst@InstallKeywords in section 10.1.

\lst@Def\langle character code \rangle\langle parameter text \rangle\langle definition \rangle
\lst@Let\langle character code \rangle\langle token \rangle

defines the specified character respectively assigns \langle token \rangle. The catcode table if not affected. Be careful if your definition has parameters: it is not safe to read more than one character ahead. Moreover, the argument can be arbitrary; sometimes it’s the next source code character, sometimes it’s some code of the listings package, e.g. \relax, \@empty, \else, \fi, and so on. Therefore don’t use TeX’s ord-operator ‘ on such an argument, e.g. don’t write \ifnum‘#1=65 to test against ‘A’.

\lst@Def and \lst@Let are relatively slow. The real definition of the standard character table differs from the following example, but it could begin with

\% \lst@Def\{9\}\{\lst@ProcessTabulator\}
\% \lst@Def\{32\}\{\lst@ProcessSpace\}
\% \lst@Def\{48\}\{\lst@ProcessDigit 0\}
\% \lst@Def\{65\}\{\lst@ProcessLetter A\}

That’s enough for the moment. Section 11 presents advanced definitions to manipulate the character table, in particular how to add new comment or string types.
9.6 On the output

The listings package uses some variables to keep the output data. Write access is not recommended. Let’s start with the easy ones.

\lst@lastother

equals \textlangle spec. token \textrangle version of the last processed nonidentifier-character. Since programming languages redefine the standard character table, we use the original \textlangle spec. token \textrangle. For example, if a double quote was processed last, \lst@lastother is not equivalent to the macro which enters and leaves string mode. It’s equivalent to \lstum", where " belongs to the control sequence. Remember that \textlangle spec. token \textrangle expands either to a printable or to a token character.

\lst@lastother is equivalent to \@empty if such a character is not available, e.g. at the beginning of a line. Sometimes an indentifier has already been printed after processing the last ‘other’ character, i.e. the character is far, far away. In this case \lst@lastother equals \relax.

\lst@outputspace

Use this predefined \textlangle spec. token \textrangle (obviously for character code 32) to test against \lst@lastother.

\lstum@backslash

Use this predefined \textlangle spec. token \textrangle (for character code 92) to test against \lst@lastother. In the replacement text for \lst@Def one could write \ifx \lst@lastother \lstum@backslash \ldots to test whether the last character has been a backslash.

\lst@SaveOutputDef{\textlangle character code \textrangle}{\textlangle macro \textrangle}

Stores the \textlangle spec. token \textrangle corresponding to \textlangle character code \textrangle in \textlangle macro \textrangle. This is the only safe way to get a correct meaning to test against \lst@lastother, for example \lst@SaveOutputDef{\textlangle character code \textrangle}{\@empty} \lstum@backslash.

You’ll get a “runaway argument” error if \textlangle character code \textrangle is not between 33 and 126 (inclusive).

Now let’s turn to the macros dealing a bit more with the output data and state.

\lst@XPrintToken

outputs the current character string and resets it. This macro keeps track of all variables described here.

token \lst@token

contains the current character string. Each ‘character’ usually expands to its printable version, but it must expand to a character token if \lst@Um is equivalent to \@empty.

counter \lst@length

is the length of the current character string.

dimension \lst@width

is the width of a single character box.
is the width of so far printed line.

global counter \lst@column

global counter \lst@pos (nonpositive)

\lst@column−\lst@pos is the length of the so far printed line. We use two counters since this simplifies tabulator handling: \lst@pos is a nonpositive representative of ‘length of so far printed line’ modulo \texttt{tabsize}. It’s usually not the biggest nonpositive representative.

\lst@CalcColumn

\@tempcnta gets \lst@column − \lst@pos + \lst@length. This is the current column number minus one, or the current column number zero based.

global dimension \lst@lostspace

equals ‘lost’ space: desired current line width minus real line width. Whenever this dimension is positive the flexible column format can use this space to fix the column alignment.

10 Package extensions

10.1 Keywords and working identifiers

The keywords aspect defines two main macros. Their respective syntax is shown on the left. On the right you’ll find examples how the package actually defines some keys.

\lst@InstallFamily

\begin{verbatim}
{(prefix)}
{(name)}
{(style name)}
{(style init)}
{(default style name)}
{(working procedure)}
{1|o}  
{d|o}  
\end{verbatim}

installs either a keyword or ‘working’ class of identifiers according to whether \texttt{(working procedure)} is empty.

The three keys \texttt{(name)}, \texttt{more(name)} and \texttt{delete(name)}, and if not empty \texttt{(style name)} are defined. The first order member of the latter one is initialized with \texttt{(style init)} if not equivalent to \texttt{relax}. If the user leaves a class style undefined, \texttt{(default style name)} is used instead. Thus, make sure that this style is always defined. In the example, the first order keywordstyle is set to \texttt{bfseries} and is the default for all other classes.

If \texttt{(working procedure)} is not empty, this code is executed when reaching such an (user defined) identifier. \texttt{(working procedure)} takes exactly one argument, namely the class number to which the actual identifier belongs to. If the code uses variables and requires values from previous calls, you must define these
variables \texttt{\textbackslash{globally}}. It’s not sure whether working procedures are executed inside a (separate) group or not.

\texttt{\textbackslash{l}} indicates a language key, i.e. the lists are reset every language selection. \texttt{\textbackslash{o}} stands for ‘other’ key. The keyword respectively working test is either installed at the \texttt{\textbackslash{DetectKeyword}} or \texttt{\textbackslash{Output}} hook according to \texttt{(d/o)}.

\texttt{\textbackslash{lst@InstallKeywords}}
\begin{verbatim}
\{(prefix)\}
\{(name)\}
\{(style name)\}
\{(style init)\}
\{(default style name)\}
\{(working procedure)\}
\langle 1|o\rangle
\langle d|o\rangle
\end{verbatim}

Same parameters, same functionality with one exception. The macro installs exactly one keyword class and not a whole family. Therefore the argument to \texttt{(working procedure)} is constant (currently empty).

The working procedure of the example reads as follows:
\begin{verbatim}
\% \{ifx\lst@lastother\lst@backslash\%
\% \let\lst@thestyle\lst@texcsstyle
\%
\% \fi\}
\end{verbatim}

What does this procedure do? First of all it is called only if a keyword from the user supplied list (or language definition) is found. The procedure now checks for a preceding backslash and sets the output style accordingly.

### 10.2 Delimiters

We describe two stages: adding a new delimiter type to an existing class of delimiters and writing a new class. Each class has its name; currently exist \texttt{Comment}, \texttt{String}, and \texttt{Delim}. As you know, the latter and the first both provide the type \texttt{l}, but there is no string which starts with the given delimiter and ends at end of line. So we’ll add it now!

First of all we extend the list of string types by
\begin{verbatim}
\% \lst@AddTo\lst@stringtypes{l}
\end{verbatim}

Then we must provide the macro which takes the user supplied delimiter and makes appropriate definitions. The command name consists of the prefix \texttt{\textbackslash{lst@}}, the delimiter name, DM for using dynamic modes, and \texttt{\textbackslash{O}} followed by the type.
\begin{verbatim}
\% \gdef\lst@StringDM@l#1#2\@empty#3#4#5{% \
\% \lst@CArg #2\relax\lst@DefDelimB{}{}{}#3{#1}{#5\lst@Lmodetrue}}
\end{verbatim}

You can put these three lines into a \texttt{.sty}-file or surround them by \texttt{\makeatletter} and \texttt{\makeatother} in the preamble of a document. And that’s all!

\begin{verbatim}
// This is a string.
This isn’t a string.
\end{verbatim}

75
You want more details, of course. Let’s begin with the arguments.

- The first argument after `\empty` is used to start the delimiter. It’s provided by the delimiter class.
- The second argument after `\empty` is used to end the delimiter. It’s also provided by the delimiter class. We didn’t need it in the example, see the explanation below.
- The third argument after `\empty` is `{⟨style⟩}{⟨start tokens⟩}`. This with a preceding `\def\lst@currstyle` is used as argument to `\lst@EnterMode`. The delimiter class also provides it. In the example we ‘extended’ #5 by `\lst@Lmodetrue` (line mode true). The mode automatically ends at end of line, so we didn’t need the end-delimiter argument.

And now for the other arguments. In case of dynamic modes, the first argument is the mode number. Then follow the user supplied delimiter(s) whose number must match the remaining arguments up to `\empty`. For non-dynamic modes, you must either allocate a static mode yourself or use a predefined mode number. The delimiters then start with the first argument.

Eventually let’s look at the replacement text of the macro. The sequence `\lst@CArg #2\relax` puts two required arguments after `\lst@DefDelimB`. The syntax of the latter macro is

```
\lst@DefDelimB
{(1st)(2nd){(rest)}}
⟨save 1st⟩
{⟨execute⟩}
{⟨delim exe modetrue⟩}
{⟨delim exe modefalse⟩}
⟨start-delimiter macro⟩
⟨mode number⟩
{⟨style⟩}{⟨start tokens⟩} {#5\lst@Lmodetrue}
defines ⟨1st⟩⟨2nd⟩⟨rest⟩ as starting-delimiter. ⟨execute⟩ is executed when the package comes to ⟨1st⟩. ⟨delim exe modetrue⟩ and ⟨delim exe modefalse⟩ are executed only if the whole delimiter ⟨1st⟩⟨2nd⟩⟨rest⟩ is found. Exactly one of them is called depending on `\lst@ifmode`. By default the package enters the mode if the delimiter is found and `\lst@ifmode` is false. Internally we make an appropriate definition of `\lst@bnext`, which can be gobbled by placing `\gobblethree` at the very end of `⟨delim exe modefalse⟩`. One can provide an own definition (and gobble the default).

⟨save 1st⟩ must be an undefined macro and is used internally to store the previous meaning of ⟨1st⟩. The arguments ⟨2nd⟩ and/or ⟨rest⟩ are empty if the delimiter has strictly less than three characters. All characters of ⟨1st⟩⟨2nd⟩⟨rest⟩ must already be active (if not empty). That’s not a problem since the macro `\lst@CArgX` does this job.
Ditto for ending-delimiter with slight differences: \(\texttt{delim exe modetrue}\) and \(\texttt{delim exe modefalse}\) are executed depending on whether \(\texttt{\lst@mode}\) equals \(\texttt{mode}\).

The package ends the mode if the delimiter is found and \(\texttt{\lst@mode}\) equals \(\texttt{mode}\). Internally we make an appropriate definition of \(\texttt{\lst@enext}\) (not \(\texttt{\lst@bnext}\)), which can be gobbled by placing \(\texttt{@gobblethree}\) at the very end of \(\texttt{delim exe modetrue}\).

\(\texttt{\lst@DefDelimBE}\)

followed by the same eight arguments as for \(\texttt{\lst@DefDelimB}\) and ... \(\texttt{end-delimiter macro}\)

This is a combination of \(\texttt{\lst@DefDelimB}\) and \(\texttt{\lst@DefDelimE}\) for the case of starting and ending delimiter being the same.

We finish the first stage by examining two easy examples. d-type strings are defined by

\[
\begin{align*}
% & \gdef\lst@StringDM@d#1#2\@empty#3#4#5\{%
% & \lst@CArg #2\relax\lst@DefDelimBE\{\}#3\{#1\}#5\}#4
\end{align*}
\]

(and an entry in the list of string types). Not a big deal. Ditto d-type comments:

\[
\begin{align*}
% & \gdef\lst@CommentDM@s#1#2#3\@empty#4#5#6\{%
% & \lst@CArg #2\relax\lst@DefDelimB\{\}#4\{#1\}#6%
% & \lst@CArg #3\relax\lst@DefDelimE\{\}#5\{#1\}
\end{align*}
\]

Here we just need to use both \(\texttt{\lst@DefDelimB}\) and \(\texttt{\lst@DefDelimE}\).

So let’s get to the second stage. For illustration, here’s the definition of the Delim class. The respective first argument to the service macro makes it delete all delimiters of the class, add the delimiter, or delete the particular delimiter only.

\[
\begin{align*}
% & \lst@Key{delim}\relax\{\lst@DelimKey\@empty\{#1\}\}
% & \lst@Key{moredelim}\relax\{\lst@DelimKey\relax\{#1\}\}
% & \lst@Key{deletedelim}\relax\{\lst@DelimKey\@nil\{#1\}\}
\end{align*}
\]

The service macro itself calls another macro with appropriate arguments.

\[
\begin{align*}
% & \gdef\lst@DelimKey#1#2\{%
% & \lst@Delim\{#2\}\relax\{Delim\}\lst@delimtypes \#1%
% & \{\lst@BeginDelim\lst@EndDelim\}
% & \i\@empty\{\lst@BeginIDelim\lst@EndIDelim\}
\end{align*}
\]

We have to look at those arguments. Above you can see the actual arguments for the Delim class, below are the Comment class ones. Note that the user supplied value covers the second and third line of arguments.
Most arguments should be clear. We'll discuss the last four. Both \{begin-and end-delim macro\} must contain exactly two control sequences, which are given to \lst@\langle\name\rangle[DM@\langle\type\rangle] to begin and end a delimiter. These are the arguments \#3 and \#4 in our first example of \lst@StringDM@l. Depending on whether the user chosen type starts with \langle\extra prefix\rangle, the first two or the last control sequences are used.

By default the package takes the delimiter(s), makes the characters active, and places them after \lst@\langle\name\rangle[DM@\langle\type\rangle]. If the user type starts with \langle\extra prefix\rangle, \langle\extra conversion\rangle might change the definition of \lst@next to choose a different conversion. The default is equivalent to \lst@XConvert with \lst@false.

Note that \langle\type\rangle never starts with \langle\extra prefix\rangle since it is discarded. The functionality must be fully implemented by choosing a different \{begin-and end-delim macro\} pair.

You might need to know the syntaxes of the \{begin-and end-delim macro\}s. They are called as follows.

\lst@Begin\langle\whatever\rangle
\begin{quote}
{\langle\mode\rangle} {\{\langle\style\rangle\} \langle\start\rangle} \langle\delimiter\rangle \@empty
\end{quote}
\lst@End\langle\whatever\rangle
\begin{quote}
{\langle\mode\rangle} \langle\delimiter\rangle \@empty
\end{quote}

The existing macros are internally defined in terms of \lst@DelimOpen and \lst@DelimClose, see the implementation.

### 10.3 Getting the kernel run

If you want new pretty-printing environments, you should be happy with section 4.5. New commands like \lstinline or \lstinputlisting are more difficult. Roughly speaking you must follow these steps.

1. Open a group to make all changes local.
2. \{Do whatever you want.\}
3. Call \lst@hk@PreSet in any case.
4. Now you might want to (but need not) use \lst@set to set some new values.
5. *(Do whatever you want.)*

6. Execute `\lst@Init` to finish initialization.

7. *(Do whatever you want.)*

8. Eventually comes the source code, which is processed by the kernel. You must ensure that the characters are either not already read or all active. Moreover you must install a way to detect the end of the source code. If you’ve reached the end, you must ...

9. ... call `\lst@DeInit` to shutdown the kernel safely.

10. *(Do whatever you want.)*

11. Close the group from the beginning.

For example, consider the `\lstinline` command in case of being not inside an argument. Then the steps are as follows.

1. `\leavevmode\bgroup` opens a group.
2. `\def\lst@boxpos{b}` ‘baseline’ aligns the listing.
3. `\lsthk@PreSet`
4. `\lstset{flexiblecolumns,#1}` (#1 is the user provided key=value list)
5. `\lsthk@TextStyle` deactivates all features not safe here.
6. `\lst@Init`
7. `\lst@Def{'}#1}{\lst@DeInit}egroup` installs the ‘end inline’ detection, where #1 is the next character after `\lstinline`. Moreover chr(13) is redefined to end the fragment in the same way but also issues an error message.

8. Now comes the source code and ...

9. ... `\lst@DeInit` (from `\lst@Def` above) ends the code snippet correctly.

11. `egroup` (also from `\lst@Def`) closes the group.

The real definition is different since we allow source code inside arguments. Read also section 18.5 if you really want to write pretty-printing commands.

## 11 Useful internal definitions

This section requires an update.
11.1 General purpose macros

\lst@AddTo\{macro\}\{(TEX\ material)\}

adds \textit{(TeX material)} globally to the contents of \texttt{(macro)}.

\lst@Extend\{macro\}\{(TEX\ material)\}

calls \lst@AddTo after the first token of \textit{(TeX material)} is \texttt{expanded after}.
For example, \lst@Extend \texttt{\a \b} merges the contents of the two macros
and stores it globally in \texttt{\a}.

\lst@lAddTo\{macro\}\{(TEX\ material)\}
\lst@lExtend\{macro\}\{(TEX\ material)\}

are local versions of \lst@AddTo and \lst@Extend.

\lst@DeleteKeysIn\{macro\}\{macro (keys to remove)\}

Both macros contain a comma separated list of keys (or keywords). All keys
appearing in the second macro are removed (locally) from the first.

\lst@ReplaceIn\{macro\}\{macro (containing replacement list)\}
\lst@ReplaceInArg\{macro\}\{(replacement list)\}

The replacement list has the form \(a_1b_1 \ldots a_nb_n\), where each \(a_i\) and \(b_i\) is a
character sequence (enclosed in braces if necessary) and may contain macros,
but the first token of \(b_i\) must not be equivalent to \texttt{\empty}. Each sequence
\(a_i\) inside the first macro is (locally) replaced by \(b_i\). The suffix \texttt{Arg}
refers to the \textit{braced} second argument instead of a (nonbraced) macro. It’s a hint that
we get the ‘real’ argument and not a ‘pointer’ to the argument.

\lst@ifSubstring\{character sequence\}\{macro\}\{(then)\}\{(else)\}

\texttt{(then)} is executed if \texttt{(character sequence)} is a substring of the contents of
\texttt{(macro)}. Otherwise \texttt{(else)} is called.

\lst@ifOneOf\{character sequence\}\texttt{relax}\{macro\}\{(then)\}\{(else)\}

\texttt{relax} terminates the first parameter here since it is faster than enclosing
it in braces. \texttt{(macro)} contains a comma separated list of identifiers. If
the character sequence is one of these indentifiers, \texttt{(then)} is executed, and
otherwise \texttt{(else)}.

\lst@Swap\{tok1\}\{tok2\}

changes places of the following two tokens or arguments \textit{without} inserting
braces. For example, \lst@Swap\{abc\}\{def\} expands to \texttt{defabc}.

\lst@ifNextChars\{macro\}\{(then)\}\{(else)\}
\lst@ifNextCharsArg\{character sequence\}\{\texttt{(then)}\}\{\texttt{(else)}\}

Both macros execute either \texttt{(then)} or \texttt{(else)} according to whether the given
character sequence respectively the contents of the given macro is found
(after the three arguments). Note an important difference between these
macros and \texttt{\empty}'s \texttt{\ifnextchar}: We remove the characters behind
the arguments until it is possible to decide which part must be executed. How-
ever, we save these characters in the macro \texttt{\eaten}, so they can be
inserted using \texttt{(then)} or \texttt{(else)}.
\lst@ifNextCharActive{(then)}{(else)}
executes (then) if next character is active, and (else) otherwise.

\lst@DefActive{macro}{character sequence}
stores the character sequence in \emph{macro}, but all characters become active. The string \emph{must not} contain a begin group, end group or escape character (\{}\{}; it may contain a left brace, right brace or backslash with other meaning (= \emph{catcode}). This command would be quite surplus if \emph{character sequence} is not already read by \TeX{} since such \emph{catcodes} can be changed easily. It is explicitly allowed that the charcaters have been read, e.g. in \texttt{\def\test{\lst@DefActive\temp{ABC}}}!

Note that this macro changes \texttt{\llccodes} \texttt{0–9} without restoring them.

\lst@DefOther{macro}{character sequence}
stores \emph{character sequence} in \emph{macro}, but all characters have catcode 12. Moreover all spaces are removed and control sequences are converted to their name without preceding backslash. For example, \texttt{\{ Chip \}} leads to \texttt{\{Chip\}} where all catcodes are 12—internally the primitive \texttt{\meaning} is used.

11.2 Character tables manipulated

\lst@SaveDef{character code}{macro}
Saves the current definition of the specified character in \emph{macro}. You should always save a character definition before you redefine it! And use the saved version instead of writing directly \lst@Process\ldots—the character could already be redefined and thus not equivalent to its standard definition.

\lst@DefSaveDef{character code}{macro}{parameter text}{definition}
\lst@LetSaveDef{character code}{macro}{token}
combine \lst@SaveDef and \lst@Def respectively \lst@Let.

Of course I shouldn’t forget to mention \emph{where} to alter the character table. Hook material at \texttt{SelectCharTable} makes permanent changes, i.e. it effects all languages. The following two keys can be used in any language definition and effects the particular language only.

\texttt{SelectCharTable=\texttt{(TeX code)}}
\texttt{MoreSelectCharTable=\texttt{(TeX code)}}
uses \texttt{(TeX code)} (additionally) to select the character table. The code is executed after the standard character table is selected, but possibly before other aspects make more changes. Since previous meanings are always saved and executed inside the new definition, this should be harmless.

Here come two rather useless examples. Each point (full stop) will cause a message ‘,’ on the terminal and in the \texttt{.log} file if language \texttt{useless} is active:
If you want to count points, you could write

\newcount\lst@points % \global
\lst@AddToHook{Init}{\global\lst@points\z@}
\lst@AddToHook{DeInit}{\message{Number of points: \the\lst@points}}
\lstdefinelanguage[2]{useless}
  {SelectCharTable=\lst@DefSaveDef{46}\lsts@point
   \global\advance\lst@points
   \lsts@point}
\global
indicates that the allocated counter is used globally. We zero the counter at the beginning of each listing, display a message about the current value at the end of a listing, and each processed point advances the counter by one.

\lst@CArg\langle active characters\rangle\relax\langle macro\rangle

The string of active characters is split into (1st), (2nd), and (rest). If one doesn’t exist, an empty argument is used. Then (macro) is called with (1st)(2nd)(rest) plus a yet undefined control sequence (save 1st). This macro is intended to hold the current definition of (1st), so (1st) can be redefined without losing information.

\lst@CArgX\langle characters\rangle\relax\langle macro\rangle

makes \langle characters\rangle active before calling \lst@CArg.

\lst@CDef\langle(1st)(2nd)(rest)\rangle\langle save 1st\rangle\{\langle execute\rangle\}\{\langle pre\rangle\}\{\langle post\rangle\}

should be used in connection with \lst@CArg or \lst@CArgX, i.e. as \langle macro\rangle there. (1st), (2nd), and (rest) must be active characters and (save 1st) must be an undefined control sequence.

Whenever the package reaches the character (1st) (in a listing), \langle execute\rangle is executed. If the package detects the whole string (1st)(2nd)(rest), we additionally execute \langle pre\rangle, then the string, and finally \langle post\rangle.

\lst@CDefX\langle(1st)(2nd)(rest)\rangle\langle save 1st\rangle\{\langle execute\rangle\}\{\langle pre\rangle\}\{\langle post\rangle\}

Ditto except that we execute \langle pre\rangle and \langle post\rangle without the original string if we reach (1st)(2nd)(rest). This means that the string is replaced by \langle pre\rangle\langle post\rangle (with preceding \langle execute\rangle).

As the final example, here’s the definition of \lst@DefDelimB.

\gdef\lst@DefDelimB#1#2#3#4#5#6#7#8{\%
 \lst@CDef\langle#1\rangle\#2\%
 \{#3\%
 \{\lst@\bnext\lst@CArgEmpty\%
 \lst@ifmode #4\else
 #\%
Implementation

12 Overture

Registers For each aspect, the required numbers of registers are listed in section 9.1 Package loading. Furthermore, the keyval package allocates one token register. The macros, boxes and counters \@temp...a/b, the dimensions \@tempdim..., and the macro \@temppa are also used, see the index.

Naming conventions Let’s begin with definitions for the user. All these public macros have lower case letters and contain lst. Private macros and variables use the following prefixes (not up-to-date?):

- \lst@ for a general macro or variable,
- \lstenv@ if it is defined for the listing environment,
- \lstsa@ for saved character meanings,
- \lsthka⟨name of hook⟩ holds hook material,
- \lst(task)@ for various kinds of keywords and working identifiers.
- \lstlanga⟨language⟩a⟨dialect⟩ contains a language and
- \lststy⟨the style⟩ contains style definition,
- \lstpatch⟨aspect⟩ to patch an aspect,
- \lsta⟨language⟩a⟨dialect⟩ contains alias,
- \lstsa⟨language⟩ contains alias for all dialects of a language,
- \lstdd⟨language⟩ contains default dialect of a language (if present).

To distinguish procedure-like macros from data-macros, the name of procedure macros use upper case letters with each beginning word, e.g. \lstAddTo. A macro with suffix @ is the main working-procedure for another definition, for example \lstMakeShortInline@ does the main work for \lstMakeShortInline.
Preamble  All files generated from this listings.dtx will get a header.

\NeedsTeXFormat{LaTeX2e}
\AtEndOfPackage{\ProvidesPackage{listings} \filedate \fileversion}

\lst@CheckVersion can be used by the various driver files to guarantee the correct version.

\ProvidesFile{lstmisc.sty} \filedate \fileversion
\lst@CheckVersion

\ProvidesFile{lstpatch.sty} \filedate \fileversion

Identification  All files will have same date and version.

What we need and who we are.

\lst@CheckVersion can be used by the various driver files to guarantee the correct version.

For example by the miscellaneous file

\ProvidesFile{lstmisc.sty}
\lst@CheckVersion

or by the dummy patch.

\ProvidesFile{lstpatch.sty}
\lst@CheckVersion
\lst@InputCatcodes \ensuremath{\textcircled{9}} and \textquoteleft \textasciitilde \textprime \textbackslash \textbackslash \textquoteleft \textcircled{1} become letters. Tabulators and EOLs are ignored; this avoids unwanted spaces—in the case I’ve forgotten a comment character.

\lst@RestoreCatcodes To load the kernel, we will change some catcodes and lccodes. We restore them at the end of package loading. Dr. Jobst Hoffmann reported an incompatibility with the \texttt{typehtml} package, which is resolved by \texttt{lccode} \\textbackslash'\textbackslash/ below.

\lst@GetAllocs \lst@ReportAllocs are used to show the allocated registers.

\lst@GetAllocs

\lst@ReportAllocs
13 General problems

All definitions in this section belong to the kernel.

13.1 Substring tests

It’s easy to decide whether a given character sequence is a substring of another string. For example, for the substring \texttt{def} we could say

\begin{verbatim}
def \lst@temp#1,\relax{% 
  \ifx \empty \empty 
    \message{the \tempcnta \string#2,} 
  \else 
    \fi 
}
def \relax 
\end{verbatim}

When \TeX passes the arguments \#1 and \#2, the second is empty if and only if \texttt{def} is not a substring. Without the additional \texttt{def\relax}, one would get a “runaway argument” error if \langle another string \rangle doesn’t contain \texttt{def}.

We use substring tests mainly in the special case of an identifier and a comma separated list of keys or keywords:

\begin{verbatim}
def \lst@temp#1.key,#2\relax{% 
  \ifx \empty \empty 
    \message{key is not a keyword} 
  \else 
    \fi 
}
def \relax 
\end{verbatim}

This works very well and is quite fast. But we can reduce run time in the case that \texttt{key} is a keyword. Then \#2 takes the rest of the string, namely all keywords after
key. Since \TeX inserts \#2 between the \@emptys, it must drop all of \#2 except the first character—which is compared with \@empty. We can redirect this rest to a third parameter:

\begin{verbatim}
\def\lst@temp#1,\#2#3\relax{% 
  \ifx \@empty\#2 \% "key" is not a keyword \else \% "key" is a keyword \fi}
\end{verbatim}

\lst@IfSubstring

That's a bit faster and an improvement for version 0.20.

\lst@IfOneOf

Ditto.

\lst@DeleteKeysIn

The submacro does the main work; we only need to expand the second macro—the list of keys to remove—and append the terminator \relax.

\lst@DeleteKeysIn

'Replacing' the very last \lst@DeleteKeysIn by \lst@RemoveCommas terminates the loop here. Note: The \@empty after \#2 ensures that this macro also works if \#2 is empty.
If we haven't reached the end of the list and if the key is not empty, we define a temporary macro which removes all appearances.

```latex
\def\lst@temp##1,#2,##2{\ifx\@empty##2\@empty\else \expandafter\lst@temp\expandafter,\fi ##2}\edef#1{\expandafter\lst@temp\expandafter,#1,#2,\@empty}\fi \fi \lst@DeleteKeysIn@#1}
```

Old definition: The following modification needs about 50% more run time. It doesn't use `\edef` and thus also works with `{` inside `#1`. However, we don't need that at the moment.

```latex
% \def\lst@temp##1,#2,##2{% % \ifx\@empty##2% % \lst@lAddTo#1{##1}% % \else % \lst@lAddTo#1{,##1}% % \expandafter\lst@temp\expandafter,% % \fi ##2}% % \let\@tempa#1\let#1\@empty % \expandafter\lst@temp\expandafter,\@tempa,#2,\@empty
```

The macro drops commas at the beginning and assigns the new value to `#1`.

```latex
\def\lst@RemoveCommas#1{\edef#1{\expandafter\lst@RC@#1\@empty}}
\def\lst@RC@#1{\ifx,#1\expandafter\lst@RC@ \else #1\fi}
```

Old definition: The following version works with `{` inside the macro `#1`.

```latex
%\def\lst@RemoveCommas#1{\expandafter\lst@RC@#1\@empty #1}%\def\lst@RC@#1{% % \ifx,#1\expandafter\lst@RC@% % \else\expandafter\lst@RC@@\expandafter#1\fi%\def\lst@RC@@#1\@empty#2{\def#2{#1}}
```

These macros are similar to `\lst@DeleteKeysIn`, except that . . .

```latex
\def\lst@ReplaceIn#1#2#3{\def\lst@ReplaceIn@#1#2#3%\lst@ReplaceInArg#1#2\@empty\@empty}
\def\lst@ReplaceInArg#1#2#3{\ifx\@empty#3\relax\else \def\lst@temp##1#2##2{% \ifx\@empty##2% \lst@lAddTo#1{##1}% \else \lst@lAddTo#1{##1#3}\expandafter\lst@temp\fi ##2} \let\@tempa#1\let#1\@empty \expandafter\lst@temp\@tempa#2\@empty \expandafter\lst@ReplaceIn@#1\fi}
```

... we replace #2 by #3 instead of ,#2, by a single comma (which removed the key #2 above).
13.2 Flow of control

\gobblethree is defined if and only if undefined.

\lst@GobbleNil

\lst@Swap is just this:

\lst@if \lst@true \lst@false

A general \if for temporary use.

\def\lst@true{\let\lst@if\iftrue}
\def\lst@false{\let\lst@if\iffalse}
\lst@false

\lst@ifNextCharsArg is quite easy: We define a macro and call \lst@ifNextChars.

\def\lst@ifNextCharsArg#1{\def\lst@tofind{#1}\lst@ifNextChars\lst@tofind}
\lst@ifNextChars

We save the arguments and start a loop.

\def\lst@ifNextChars{\expandafter\lst@ifNextChars@\lst@tofind\relax}

Expand the characters we are looking for.

\def\lst@ifNextChars@{\expandafter\lst@ifNextChars@@\lst@tofind\relax}

Now we can refine \lst@tofind and append the input character #3 to \lst@eaten.

\def\lst@ifNextChars@@#1#2\relax#3{\def\lst@tofind{#2}\lst@eaten\lst@tofind{#3}\if\\lst@tofind\relax\lst@eaten\lst@tofind{#3}\else\if\lst@tofind\relax\lst@eaten\lst@tofind{#3}\else\lst@next#3\fi\fi}

\lst@ifNextCharActive We compare the character #3 with its active version \lowercase{~}. Note that the right brace between \if\ and #3 ends the \lowercase. The \endgroup restores the \lccode.

\def\lst@ifNextCharActive#1#2#3{%\begingroup \lccode'~='\endgroup \lowercase\\hfuzz=\if\lst@tofind\relax\lst@tofind\else#3\fi}
\lst@for A for-loop with expansion of the loop-variable. This was improved due to a suggestion by Hendri Adriaens.
\begin{verbatim}
166 \def\lst@for#1\do#2{\%
167 \def\lst@forbody##1{#2}\%
168 \def\@tempa{#1}\%
169 \ifx\@tempa\@empty\else\expandafter\lst@f@r#1,\@nil,\fi
170 }\%
171 \def\lst@f@r#1,{% 
172 \def\@tempa{#1}\%
173 \ifx\@tempa\@nnil\else\lst@forbody{#1}\expandafter\lst@f@r\fi
174 }
\end{verbatim}

13.3 Catcode changes
A character gets its catcode right after reading it and \TeX has no primitive command to change attached catcodes. However, we can replace these characters by characters with same ASCII codes and different catcodes. It’s not the same but suffices since the result is the same. Here we treat the very special case that all characters become active. If we want \lst@arg to contain an active version of the character \#1, a prototype macro could be
\begin{verbatim}
\def\lst@MakeActive#1{\lccode'\^\^@='\^\^@#1\lowercase{\def\lst@arg{\^\^@}}}}
\end{verbatim}
The \lowercase changes the ASCII code of ~ to the one of \#1 since we have said that \#1 is the lower case version of ~. Fortunately the \lowercase doesn’t change the catcode, so we have an active version of \#1. Note that ~ is usually active.

\lst@MakeActive We won’t do this character by character. To increase speed we change nine characters at the same time (if nine characters are left).

To do: This was introduced when the delimiters were converted each listings. Now this conversion is done only each language selection. So we might want to implement a character by character conversion again to decrease the memory usage.

We get the argument, empty \lst@arg and begin a loop.
\begin{verbatim}
175 \def\lst@MakeActive#1{% 
176 \let\lst@temp\@empty \lst@MakeActive@#1%
177 \relax\relax\relax\relax\relax\relax\relax\relax\relax\relax\relax
\end{verbatim}
There are nine \relaxes since \lst@MakeActive@ has nine parameters and we don’t want any problems in the case that \#1 is empty. We need nine active characters now instead of a single ~. We make these catcode changes local and define the coming macro \global.

First we \let the next operation be \relax. This aborts our loop for processing all characters (default and possibly changed later). Then we look if we have at least one character. If this is not the case, the loop terminates and all is done.
\begin{verbatim}
182 \gdef\lst@MakeActive@#1#2#3#4#5#6#7#8#9{%\let\lst@next\relax
183 \ifx#1\relax
184 \else \lccode'\^\^@='\^\^@#1%
\end{verbatim}
Otherwise we say that `^@=chr(0)` is the lower case version of the first character. Then we test the second character. If there is none, we append the lower case `^@` to `\lst@temp`. Otherwise we say that `^A=chr(1)` is the lower case version of the second character and we test the next argument, and so on.

\begin{verbatim}
  \ifx#2\relax
  \lowercase{\lst@lAddTo\lst@temp{^@}}%
  \else \lccode'\^A='#2%
  \fi
\end{verbatim}

\begin{verbatim}
  \ifx#3\relax
  \lowercase{\lst@lAddTo\lst@temp{^@^A}}%
  \else \lccode'\^B='#3%
  \fi
\end{verbatim}

\begin{verbatim}
  \ifx#4\relax
  \lowercase{\lst@lAddTo\lst@temp{^@^A^B}}%
  \else \lccode'\^C='#4%
  \fi
\end{verbatim}

\begin{verbatim}
  \ifx#5\relax
  \lowercase{\lst@lAddTo\lst@temp{^@^A^B^C}}%
  \else \lccode'\^D='#5%
  \fi
\end{verbatim}

\begin{verbatim}
  \ifx#6\relax
  \lowercase{\lst@lAddTo\lst@temp{^@^A^B^C^D}}%
  \else \lccode'\^E='#6%
  \fi
\end{verbatim}

\begin{verbatim}
  \ifx#7\relax
  \lowercase{\lst@lAddTo\lst@temp{^@^A^B^C^D^E}}%
  \else \lccode'\^F='#7%
  \fi
\end{verbatim}

\begin{verbatim}
  \ifx#8\relax
  \lowercase{\lst@lAddTo\lst@temp{^@^A^B^C^D^E^F}}%
  \else \lccode'\^G='#8%
  \fi
\end{verbatim}

\begin{verbatim}
  \lowercase{\lst@lAddTo\lst@temp{^@^A^B^C^D^E^F^G}}%
\end{verbatim}

If nine characters are present, we append (lower case versions of) nine active characters and call this macro again via redefining `\lst@next`.

\begin{verbatim}
  \lowercase{\lst@lAddTo\lst@temp{^@^A^B^C^D^E^F^G^H}}%
  \let\lst@next\lst@MakeActive@
\end{verbatim}

\begin{verbatim}
\endgroup
\end{verbatim}

This `\endgroup` restores the catcodes of `chr(0)`–`chr(8)`, but not the catcodes of the characters inside `\lst@MakeActive@` since they are already read.

Note: A conversion from an arbitrary ‘catcode–character code’ table back to \TeX’s catcodes is possible if we test against the character codes (either via `\ifnum` or `\ifcase`). But control sequences and begin and end group characters definitely need some special treatment. However I haven’t checked the details. So just ignore this and don’t bother me for this note. : – )

\lst@DefActive An easy application of `\lst@MakeActive`.

\begin{verbatim}
\def\lst@DefActive#1#2{\lst@MakeActive{#2}\let#1\lst@temp}
\end{verbatim}

\lst@DefOther We use the fact that `\meaning` produces catcode 12 characters except spaces stay spaces. `\escapechar` is modified locally to suppress the output of an escape character. Finally we remove spaces via \LaTeX’s `\zap@space`, which was proposed by Rolf Nieprasch—not in this context, but that doesn’t matter.

\begin{verbatim}
\def\lst@DefOther#1#2{\zap@space}
\end{verbatim}
13.4 Applications to 13.3

If an environment is used inside an argument, the listing is already read and we can do nothing to preserve the catcodes. However, under certain circumstances the environment can be used inside an argument—that’s at least what I’ve said in the User’s guide. And now I have to work for it coming true. Moreover we define an analogous conversion macro for the `fancyverb` mode.

\lst@InsideConvert\{}\langle\text{T\TeX\ material\ (already\ read)}\rangle\}\{\text{verbatim}\}\{\text{active}\}

appends a verbatim version of the argument to `\lst@arg`, but all appended characters are active. Since it’s not a character to character conversion, ‘verbatim’ needs to be explained. All characters can be typed in as they are except \, \{, \} and %. If you want one of these, you must write \, \{, \} and \% instead. If two spaces should follow each other, the second (third, fourth, . . . ) space must be entered with a preceding backslash.

\lst@XConvert\{}\langle\text{T\TeX\ material\ (already\ read)}\rangle\}\{\text{verbatim}\}\{\text{active}\}

appends a ‘verbatim’ version of the argument to `\lst@arg`. Here T\TeX\ material is allowed to be put inside argument braces like \{(*}\{*)\}. The contents of these arguments are converted, the braces stay as curly braces.

If `\lst@if` is true, each second argument is treated differently. Only the first character (of the delimiter) becomes active.

\lst@InsideConvert\{\text{verbatim}\}\{\text{active}\}

If `mathescape` is not on, we call (near the end of this definition) a submacro similar to `\zap@space` to replace single spaces by active spaces. Otherwise we check whether the code contains a pair `\$\ldots\$` and call the appropriate macro.
Finally we end the \lowercase and close a group.}

The next definition has been used above to check for $...$ and the following one keeps the math contents from being converted. This feature was requested by Dr. Jobst Hoffmann.

\def\lst@InsideConvert@e#1$#2\@nil{% 
  \ifx\@empty#2\@empty \lst@false \else \lst@true \fi}
\def\lst@InsideConvert@ey#1$#2$#3\@nil{% 
  \lst@InsideConvert@#1 \@empty 
  \lst@lAddTo\lst@arg{
    \lst@ifdropinput\else
    \lst@TrackNewLines\lst@OutputLostSpace \lst@XPrintToken 
    \setbox\@tempboxa=\hbox\bgroup\lst@escapebegin 
    #2% 
    \lst@escapeend\egroup \lst@CalcLostSpaceAndOutput 
    \lst@whitespacefalse 
    \fi}\
  \def\lst@next{\lst@InsideConvert{#3}}%}
\def\lst@XConvert{\@ifnextchar\bgroup \lst@XConvertArg \lst@XConvert@}

Having no \bgroup, we look whether we’ve found the end of the input, and convert one token ((non)active character or control sequence) and continue.

\def\lst@XConvertArg#1{% 
  {\lst@false \let\lst@arg\@empty 
  \lst@XConvert#1\@nil 
  \global\let\@gtempa\lst@arg}\
  \lst@lExtend\lst@arg{\expandafter{\@gtempa}}\
  \expandafter\lst@XConvertNext}
\def\lst@XConvertNext{% 
  \lst@if \expandafter\lst@XConvertX 
  \else \expandafter\lst@XConvert \fi}
\def\lst@XConvertX#1{\
  \ifx\@nil#1\else 
  \lst@XConvertX@#1\relax 
  \expandafter\lst@XConvert 
  \fi}
\def\lst@XConvertX@#1#2\relax{...}

Now we make only the first character active.

\def\lst@XConvertX#1{% 
  \ifx\@nil#1\else 
    \begingroup\lccode’\~='\#1\lowercase{\endgroup 
  \lst@lAddTo\lst@arg{\expandafter{\@gtempa}}% 
  \expandafter\lst@XConvertNext 
  \fi}
\def\lst@XConvertNext{% 
  \lst@if \expandafter\lst@XConvertX 
  \else \expandafter\lst@XConvert \fi}

\def\lst@XConvertX#1{% 
  \ifx\@nil#1\else 
    \lst@XConvertX@#1\relax 
  \expandafter\lst@XConvert 
  \fi}
13.5 Driver file handling*

The listings package is split into several driver files, miscellaneous (= aspect) files, and one kernel file. All these files can be loaded partially and on demand—except the kernel which provides this functionality.

\lst@Require\{⟨name⟩\}{⟨prefix⟩}{⟨feature list⟩}{⟨file list macro⟩}

tries to load all items of ⟨feature list⟩ from the files listed in ⟨file list macro⟩. Each item has the form \[[⟨sub⟩]⟨feature⟩\]. \lst@if equals \iftrue if and only if all items were loadable.

The macro ⟨alias⟩ gets an item as argument and must define appropriate versions of \lst@oalias and \lst@malias. In fact the feature associated with these definitions is loaded. You can use ⟨alias⟩=\lst@NoAlias for no substitution.

⟨prefix⟩ identifies the type internally and ⟨name⟩ is used for messages.

For example, \lstloadaspects uses the following arguments where \#1 is the list of aspects: \{aspects\}a{\#1}\lst@NoAlias\lstaspectfiles.

\lst@DefDriver\{⟨name⟩\}{⟨prefix⟩}{⟨interface macro⟩}\iftrue|false
\lst@ifRequired[⟨sub⟩]{⟨feature⟩}{⟨then⟩}{⟨else⟩}

is used inside a driver file by the aspect, language, or whatever else defining commands. ⟨then⟩ is executed if and only if \{(sub)\}{⟨feature⟩} has been requested via \lst@Require. Otherwise ⟨else⟩ is executed—which is also the case for subsequent calls with the same \{(sub)\}{⟨feature⟩}.

⟨then⟩ and ⟨else⟩ may use \lst@prefix (read access only).

\lst@BeginAspect in section 13.6 and \lst@DefDriver serve as examples.
Init things and input files if and as long as it is necessary.

\global\let\lst@loadaspects\@empty
\lst@InputCatcodes
\ifx\lst@require\@empty\else
\lst@for\#5\do{\%\lst@require\@empty\else
\InputIfFileExists{##1}{}{}\%
\fi}\%
\fi

Issue error and call \lst@false (after closing the local group) if some items weren’t loadable.

\ifx\lst@require\@empty\else\PackageError{Listings}{Couldn’t load requested #1}\%
{The following #1s weren’t loadable: `\@spaces \lst@require` This may cause errors in the sequel.}\%
\aftergroup\lst@false\fi

Request aspects.

\ifx\lst@loadaspects\@empty\else\lst@RequireAspects\lst@loadaspects\fi
\fi\endgroup

\lst@IfRequired uses \lst@IfOneOf and adds some code to (then) part: delete the now loaded item from the list and define \lst{(prefix)@\#(feature)$\langle sub \rangle}$.

\def\lst@IfRequired[\#1]\#2{\lst@NormedDef\lst@temp{\[#1\] \#2}\%
\expandafter\lst@IfRequired\lst@temp\relax}
\def\lst@IfRequired\[#1\] \#2\relax\#3{\lst@IfOneOf \#2$\#1\relax\lst@require
\{\lst@DeleteKeysIn\lst@require\#2$\#1,\relax%,
\global\expandafter\let\csname\@lst\lst@prefix @\#2$\#1\endcsname\@empty
\csname\@lst\lst@prefix @\#2$\#1\endcsname\#3\}}

\lst@require
\let\lst@require\@empty

\lst@NoAlias just defines \lst@oalias and \lst@malias.

\def\lst@NoAlias[\#1]\#2{\lst@NormedDef\lst@oalias{\#1}\lst@NormedDef\lst@malias{\#2}}

\lst@LAS just defines \lst@hasalias and \lst@msalias.

\def\lst@hasalias\#1\#2\#3\#4\#5\#6\#7{\%\lst@Require\#1\{\#2\}\{\#3\}\#4\#5\%\#4\#3\%
\lst@DeleteKeysIn\lst@require\#2$\#1,\relax%,
\global\expandafter\let\csname\@lst\lst@prefix @\#2$\#1\endcsname\@empty
\csname\@lst\lst@prefix @\#2$\#1\endcsname\#3\}}

\lst@require
\let\lst@require\@empty

\lst@NoAlias just defines \lst@oalias and \lst@malias.

\def\lst@NoAlias[\#1]\#2{\lst@NormedDef\lst@oalias{\#1}\lst@NormedDef\lst@malias{\#2}}

\lst@LAS
\lst@RequireAspects make use of the just developed definitions.
\lstloadaspects
\lstaspectfiles This macro is defined if and only if it's undefined yet.
\lst@DefDriver Test the next character and reinsert the arguments.
\lst@DefDriver@ This macro is defined if and only if it's undefined yet.
\lst@DefDriver@@ We set \lst@if locally true if the item has been requested.
\lst@XXDefDriver We get the aspect argument, and (if not empty) load the aspects immediately if called by a public or internal command.
\lst@XXDefDriver We insert an additional 'also' key=value pair.

13.6 Aspect commands
This section contains commands used in defining 'lst-aspects'.

\lst@RequireAspects \lstloadaspects \lstaspectfiles
\lst@UserCommand is mainly equivalent to \gdef.

\lst@BeginAspect A straight-forward implementation:

\newcommand*{\lst@BeginAspect}[2][]{%
\def\lst@curraspect{#2}%
\ifx \lst@curraspect\@empty
\expandafter\lst@GobbleAspect
\else
If ⟨aspect name⟩ is not empty, there are certain other conditions not to define the aspect (as described in section 9.2).
\let\lst@next\@empty
\def\lst@next{%
\message{^^JDefine lst-aspect '#2':}
\lst@GetAllocs%
\lst@IfRequired[{}]{#2}%
\{\lst@RequireAspects[{}]
\lst@if\else \let\lst@next\lst@GobbleAspect \fi}%
\expandafter\lst@next%
}\fi
%
\centering
\lst@EndAspect finishes an aspect definition.

\def\lst@EndAspect{%
c\csname\@lst patch@\lst@curraspect\endcsname
\lst@ReportAllocs
\let\lst@curraspect\@empty}

\lst@GobbleAspect drops all code up to the next \lst@EndAspect.

\long\def\lst@GobbleAspect#1\lst@EndAspect{%
\let\lst@curraspect\@empty}

\lst@Key The command simply defines the key. But we must take care of an optional parameter and the initialization argument #2.

\def\lst@Key#1#2{%
\message{#1,}
\@ifnextchar[{
lstKV@def{#1}{#2}}%
{\def\lst@temp{\lst@Key@{#1}{#2}}
\afterassignment\lst@temp
\global\@namedef{KV@\@lst @#1}##1}}

Now comes a renamed and modified copy from a keyval macro: We need global key definitions.

\def\lstKV@def#1#2[#3]{%
\global\@namedef{KV@\@lst @#1@[#1]#1\#2\#3}}

We initialize the key if the first token of #2 is not \relax.

\def\lst@Key@#1#2{%
\ifx\relax#2\@empty\else
\begingroup \globaldefs\@ne
\csname KV@\@lst @#1\endcsname\#1\endcsname
\def\lst@temp{\lst@Key@{#1}{#2}}
\afterassignment\lst@temp
\global\@namedef{KV@\@lst @#1\#1}\relax
\endgroup}

We initialize the key if the first token of #2 is not \relax.

\def\lst@Key@#1#2{%
\ifx\relax#2\@empty\else
\begingroup \globaldefs\@ne
\csname KV@\@lst @#1\endcsname\#1\endcsname
\def\lst@temp{\lst@Key@{#1}{#2}}
\afterassignment\lst@temp
\global\@namedef{KV@\@lst @#1\#1\#1}\relax
\endgroup

97
\lst@UseHook is very, very, \ldots, very (hundreds of times) easy.
\def\lst@UseHook#1{\csname\@lst hk@#1\endcsname}

All use the same submacro.
\def\lst@AddToHook{\lst@ATH@\iffalse\lst@AddTo}
\def\lst@AddToHookExe{\lst@ATH@\iftrue\lst@AddTo}
\def\lst@AddToHookAtTop{\lst@ATH@\iffalse\lst@AddToAtTop}

If and only if the boolean value is true, the hook material is executed globally.
\long\def\lst@ATH@#1#2#3#4{%} %
\@ifundefined{\@lst hk@#3}{%}
\expandafter\gdef\csname\@lst hk@#3\endcsname{}}{}%
\expandafter#2\csname\@lst hk@#3\endcsname{#4}%
\def\lst@temp{#4}%
#1% \iftrue|false
\begingroup \globaldefs\@ne \lst@temp \endgroup \fi}

\lst@AddTo Note that the definition is global!
\long\def\lst@AddTo#1#2{%}
\expandafter\gdef\expandafter#1\expandafter{#1#2}

\lst@AddToAtTop We need a couple of \texttt{expandafter}s now. Simply note that we have \\texttt{\textbackslash expandafter}\texttt{\textbackslash expandafter}\texttt{\textbackslash expandafter}\texttt{\textbackslash expandafter} after the ‘first phase’ of expansion.
\def\lst@AddToAtTop#1#2{%}
\expandafter\expandafter\expandafter\gdef
\expandafter\expandafter\expandafter#1\expandafter{\expandafter\lst@temp#1}

\lst@lAddTo A local version of \texttt{\lst@AddTo} \ldots
\def\lst@lAddTo#1#2{%}
\expandafter\def\expandafter#1\expandafter{#1#2}

\lst@Extend \ldots and here we expand the first token of the second argument first.
\def\lst@Extend#1\#2{%}
\expandafter\lst@Extend\expandafter#1\expandafter{\expandafter#2}
\def\lst@lExtend#1\#2{%}
\expandafter\lst@lExtend\expandafter#1\expandafter{\expandafter#2}

To do: This should never be changed to
\% \texttt{\def\lst@Extend\#1{%}
\texttt{\expandafter\lst@Extend\texttt{\textbackslash expandafter}\texttt{\textbackslash expandafter}}
\% \texttt{\def\lst@lExtend\#1{%}
\texttt{\expandafter\lst@lExtend\texttt{\textbackslash expandafter}}}
13.7 Interfacing with keyval

The keyval package passes the value via the one and only parameter \#1 to the definition part of the key macro. The following commands may be used to analyse the value. Note that we need at least version 1.10 of the keyval package. Note also that the package removes a naming conflict with AMS classes—reported by Ralf Quast.

\RequirePackage{keyval}[1997/11/10]

\lstKV@TwoArg Define temporary macros and call with given arguments \#1. We add empty arguments for the case that the user doesn’t provide enough.
\lstKV@ThreeArg
\lstKV@FourArg

\def\lstKV@TwoArg#1#2{%\gdef\@gtempa##1##2{#2}\@gtempa#1{}{}\}
\def\lstKV@ThreeArg#1#2{%\gdef\@gtempa##1##2##3{#2}\@gtempa#1{}{}{}\}
\def\lstKV@FourArg#1#2{%\gdef\@gtempa##1##2##3##4{#2}\@gtempa#1{}{}{}{}\}

There’s one question: What are the global definitions good for? \lst@Key might set \globaldefs to one and possibly calls this macro. That’s the reason why we use global definitions here and below.

\lstKV@OptArg We define the temporary macro \@gtempa and insert default argument if necessary.
\lstKV@OptArg@#1#2\@\)
\lstKV@OptArg@@#1#2\@\)

\def\lstKV@OptArg\[#1\]#2#3{%\gdef\@gtempa[#1]##2{#3}\lstKV@OptArg@{#1}#2\@\}
\def\lstKV@OptArg@#1{%\@ifnextchar[\lstKV@OptArg@@{#1}\]
\def\lstKV@OptArg@@#1#2\@\){\@gtempa[#1]{#2}}

\lstKV@XOptArg Here \#3 is already a definition with at least two parameters whose first is enclosed in brackets.
\lstKV@CSTwoArg Just define temporary macro and call it.
\lstKV@SetIf We simply test the lower case first character of \#1.
\lstKV@SwitchCases is implemented as a substring test. The original version used an &, which produced a bug—see p. 65.
Finally this main user interface macro. We change catcodes for reading the argument.

\lst@UserCommand{\lstset{\begin{group} \lst@setcatcodes \lstset\end{group}\lstset@}}
\def\lstset@#1{\endgroup \ifx\@empty#1\@empty\else\setkeys{lst}{#1}\fi}

\lst@setcatcodes contains all catcode changes for \lstset. The equal-sign has been added after a bug report by Bekir Karaoglu—babel’s active equal sign clashes with keyval’s usage. \catcode`\"=12\relax has been removed after a bug report by Heiko Bauke—hopefully this introduces no other bugs.
\def\lst@setcatcodes{\makeatletter \catcode`\==12\relax}

To do: Change more catcodes?

13.8 Internal modes

\lst@NewMode We simply use \chardef for a mode definition. The counter \lst@mode mainly keeps the current mode number. But it is also used to advance the number in the macro \lst@newmode—we don’t waste another counter.
\def\lst@NewMode#1{%\ifx\@undefined#1%\lst@mode\lst@newmode\relax \advance\lst@mode\@ne\xdef\lst@newmode{\the\lst@mode}% \global\chardef#1=\lst@mode\lst@mode\lst@nomode\fi}
\lst@mode\lst@nomode

\lst@NewMode We allocate the counter and the first mode.
\newcount\lst@mode
\def\lst@newmode{\m@ne}% init
\lst@NewMode\lst@nomode % init (of \lst@mode :-)

\lst@UseDynamicMode For dynamic modes we must not use the counter \lst@mode (since possibly already valued). \lst@dynamicmode substitutes \lst@newmode and is a local definition here, ...
\def\lst@UseDynamicMode{%\@tempcnta\lst@dynamicmode\relax \advance\@tempcnta\@ne\edef\lst@dynamicmode{\the\@tempcnta}\expandafter\lst@Swap\expandafter{\expandafter{\lst@dynamicmode}}}

\lst@UseDynamicMode initialized each listing with the current ‘value’ of \lst@newmode.
\lst@AddToHook{InitVars}{\let\lst@dynamicmode\lst@newmode}

\lst@EnterMode Each mode opens a group level, stores the mode number and execute mode specific tokens. Moreover we keep all these changes in mind (locally) and adjust internal variables if the user wants it.
\def\lst@EnterMode#1#2{%\bgroup \lst@mode=#1\relax #2\lst@FontAdjust\lst@lAddTo\lst@entermodes{\lst@EnterMode{#1}{#2}}\lst@AddToHook{InitVars}{\let\lst@entermodes\@empty}\let\lst@entermodes\@empty % init

The initialization has been added after a bug report from Herfried Karl Wagner.
\lst@LeaveMode  We simply close the group and call \lsthk@EndGroup if and only if the current mode is not \lst@nomode.
\def\lst@LeaveMode{%
  \ifnum\lst@mode=\lst@nomode\else
    \egroup \expandafter\lsthk@EndGroup
  \fi}
\lst@AddToHook{EndGroup}{}% init

\lst@InterruptModes  We put the current mode sequence on a stack and leave all modes.
\def\lst@InterruptModes{%
  \lst@Extend\lst@modestack{\expandafter{\lst@entermodes}}%
  \lst@LeaveAllModes
\lst@AddToHook{InitVars}{\global\let\lst@modestack\@empty}

\lst@ReenterModes  If the stack is not empty, we leave all modes and pop the topmost element (which is the last element of \lst@modestack).
\def\lst@ReenterModes{%
  \ifx\lst@modestack\@empty\else
    \lst@LeaveAllModes
    \global\let\@gtempa\lst@modestack
    \global\let\lst@modestack\@empty
    \expandafter\lst@ReenterModes@\@gtempa\relax
  \fi}
\def\lst@ReenterModes@#1#2{%
  \ifx\relax#2\@empty
    If we've reached \relax, we've also found the last element: we execute \#1 and gobble \#2=\relax after \fi.
  \gdef\@gtempa##1{#1}%
  \expandafter\@gtempa
  \else
    Otherwise we just add the element to \lst@modestack and continue the loop.
  \lst@AddTo\lst@modestack{#1}%
  \expandafter\lst@ReenterModes@{#2}}
\lst@LeaveAllModes  Leaving all modes means closing groups until the mode equals \lst@nomode.
\def\lst@LeaveAllModes{%
  \ifnum\lst@mode=\lst@nomode
    \expandafter\lsthk@EndGroup
  \else
    \expandafter\egroup\expandafter\lst@LeaveAllModes
  \fi}
\lst@AddToHook{ExitVars}{\lst@LeaveAllModes}

\lst@Pmode, \lst@GPmode  The 'processing' and the general purpose mode.
\lst@NewMode\lst@Pmode
\lst@NewMode\lst@GPmode

101
The usual macro to value a boolean except that we also execute a hook.
\[\texttt{\begin{verbatim}
def \lst@modetrue{\let \lst@ifmode \iftrue \lsthk@ModeTrue}
def \lst@ifmode{\iffalse % init}
def \lst@AddToHook{ModeTrue}{}% init
\end{verbatim}}\]
Comment lines use a static mode. It terminates at end of line.
\[\texttt{\begin{verbatim}
def \lst@Lmodetrue{\let \lst@ifLmode \iftrue}
def \lst@ifLmode{\iffalse % init}
def \lst@AddToHook{EOL}{\@whilesw \lst@ifLmode\fi \lst@LeaveMode}
\end{verbatim}}\]

13.9 Diverse helpers
\[\texttt{\begin{verbatim}
def \lst@NormedDef{\lowercase{\edef #1{\zap@space\@empty}}}
def \lst@NormedNameDef{\lowercase{\edef \lst@temp{\zap@space#1\@empty}%%}
\expandafter\xdef\csname \lst@temp\endcsname{\zap@space#2\@empty}}}
\end{verbatim}}\]

14 Doing output
14.1 Basic registers and keys
\[\texttt{\begin{verbatim}
def \lst@token\newtoks \lst@token \newcount \lst@length
\end{verbatim}}\]
The two registers get empty respectively zero at the beginning of each line. After receiving a report from Claus Atzenbeck—I removed such a bug many times—I decided to reset these registers in the `EndGroup` hook, too.

535 \def\lst@ResetToken{\lst@token{}\lst@length\z@}
536 \lst@AddToHook{InitVarsBOL}{\lst@ResetToken \let\lst@lastother\@empty}
537 \lst@AddToHook{EndGroup}{\lst@ResetToken \let\lst@lastother\@empty}

The macro \lst@lastother will be equivalent to the last ‘other’ character, which leads us to \lst@ifletter.

\lst@ifletter indicates whether the token contains an identifier or other characters.

538 \def\lst@lettertrue{\let\lst@ifletter\iftrue}
539 \def\lst@letterfalse{\let\lst@ifletter\iffalse}
540 \lst@AddToHook{InitVars}{\lst@letterfalse}

\lst@Append puts the argument into the output queue.

541 \def\lst@Append#1{\advance\lst@length\@ne
542 \lst@token=\expandafter{\the\lst@token#1}}

\lst@AppendOther Depending on the current state, we first output the character string as an identifier. Then we save the ‘argument’ via \futurelet and call the macro \lst@Append to do the rest.

543 \def\lst@AppendOther{%
544 \lst@ifletter \lst@Output\lst@letterfalse \fi
545 \futurelet\lst@lastother\lst@Append}

\lst@AppendLetter We output a non-identifier string if necessary and call \lst@Append.

546 \def\lst@AppendLetter{%
547 \lst@ifletter\else \lst@OutputOther\lst@lettertrue \fi
548 \lst@Append}

\lst@SaveToken \lst@RestoreToken If a group end appears and ruins the character string, we can use these macros to save and restore the contents. \lst@thestyle is the current printing style and must be saved and restored, too.

549 \def\lst@SaveToken{%
550 \global\let\lst@gthestyle\lst@thestyle
551 \global\let\lst@glastother\lst@lastother
552 \xdef\lst@RestoreToken{\noexpand\lst@token{\the\lst@token}%
553 \noexpand\lst@length\the\lst@length\relax
554 \noexpand\let\noexpand\lst@thestyle
555 \noexpand\lst@gthestyle
556 \noexpand\let\noexpand\lst@lastother
557 \noexpand\lst@glastother}}

Now – that means after a bug report by Rolf Niepraschk – \lst@lastother is also saved and restored.

\lst@ifLastOtherOneOf Finally, this obvious implementation.

558 \def\lst@ifLastOtherOneOf#1{\lst@ifLastOtherOneOf@ #1\relax}
559 \def\lst@ifLastOtherOneOf@#1{%
560 \ifx #1\relax
561 \expandafter\@secondoftwo
562 \else
The current position is either the dimension \lst@currlwidth, which is the horizontal position without taking the current character string into account, or it’s the current column starting with number 0. This is \lst@column − \lst@pos + \lst@length. Moreover we have \lst@lostspace which is the difference between the current and the desired line width. We define macros to insert this lost space.

\lst@currlwidth the current line width and two counters.
\lst@column \lst@pos \lst@CalcColumn sets \@tempcnta to the current column. Note that \lst@pos will be nonpositive.
\lst@lostspace Whenever this dimension is positive we can insert space. A negative ‘lost space’ means that the printed line is wider than expected.
\lst@UseLostSpace We insert space and reset it if and only if \lst@lostspace is positive.
\lst@InsertLostSpace Ditto, but insert even if negative. \lst@Kern will be defined very soon.

Column widths Here we deal with the width of a single column, which equals the width of a single character box. Keep in mind that there are fixed and flexible column formats.

\lst@width \basewidth assigns the values to macros and tests whether they are negative.
We set the dimension in a special hook.

\lst@AddToHook{FontAdjust}{
\lst@width=\lst@ifflexible\lst@widthflexible
\else\lst@widthfixed\fi \relax}

This hook is controlled by a switch and is always executed at InitVars.

\lst@Key{fontadjust}{false}[t]{\lstKV@SetIf{#1}\lst@iffontadjust}
\def\lst@FontAdjust{\lst@iffontadjust \lsthk@FontAdjust \fi}
\lst@AddToHook{InitVars}{\lsthk@FontAdjust}

\section*{14.2 Low- and mid-level output}

\textbf{Doing the output} means putting the character string into a box register, updating all internal data, and eventually giving the box to \TeX.

\lst@OutputBox (The lowest level is the output of a box register. Here we use \texttt{\box#1} as argument to \texttt{\lst@alloverstyle}.

\def\lst@OutputBox#1{\lst@alloverstyle{\box#1}}

Alternative: Instead of \texttt{\global\advance\lst@currlwidth \wd⟨box number⟩} in both definitions \texttt{\lst@Kern} and \texttt{\lst@CalcLostSpaceAndOutput}, we could also advance the dimension here. But I decided not to do so since it simplifies possible redefinitions of \texttt{\lst@OutputBox}: we need not to care about \texttt{\lst@currlwidth}.

\def\lst@alloverstyle#1{% init
\lst@Kern (has been used to insert `lost space'. It must not use \texttt{@tempboxa} since that ...\def\lst@Kern#1{%\setbox\z@\hbox{{\lst@currstyle{\kern#1}}}\global\advance\lst@currlwidth \wd\z@\global\advance\lst@lostspace \lst@length\lst@width\global\advance\lst@lostspace-\wd\@tempboxa\global\advance\lst@currlwidth \wd\@tempboxa\global\advance\lst@pos \lst@length\global\advance\l@pos \lst@length\global\advance\l@pos \lst@pos - \lst@length\def\lst@CalcLostSpaceAndOutput{%\global\advance\lst@lostspace \lst@length\lst@width\global\advance\lst@lostspace-\wd\@tempboxa\global\advance\lst@currlwidth \wd\@tempboxa\global\advance\lst@pos \lst@length\global\advance\l@pos \lst@length\global\advance\l@pos \lst@pos \lst@length
\lst@CalcLostSpaceAndOutput (... is used here. We keep track of \texttt{\lst@lostspace}, \texttt{\lst@currlwidth} and \texttt{\lst@pos}.\def\lst@CalcLostSpaceAndOutput{%\global\advance\lst@lostspace \lst@length\lst@width\global\advance\lst@lostspace-\wd\@tempboxa\global\advance\lst@currlwidth \wd\@tempboxa\global\advance\lst@pos \lst@length\global\advance\l@pos \lst@length\global\advance\l@pos \lst@pos \lst@length

Before \texttt{@tempboxa} is output, we insert space if there is enough lost space. This possibly invokes \texttt{\lst@Kern} via ‘insert half lost space’, which is the reason for why we mustn’t use \texttt{@tempboxa} above. By redefinition we prevent \texttt{\lst@OutputBox} from using any special style in \texttt{\lst@Kern}.\lst@Kern%}
Finally we can output the new box.

\lst@OutputBox\@tempboxa \lsthk@PostOutput
\lst@AddToHook{PostOutput}{% init
\lst@OutputToken

Now comes a mid-level definition. Here we use \lst@token to set \@tempboxa and eventually output the box. We take care of font adjustment and special output styles. Yet unknown macros are defined in the following subsections.

\def\lst@OutputToken{%
\lst@TrackNewLines \lst@OutputLostSpace
\lst@ifgobbledws
\lst@gobbledwhitespacefalse
\lst@discretionary
\fi
\lst@CheckMerge
\lst@thestyle{\lst@FontAdjust}
\setbox\@tempboxa{\lsthk@OutputBox
\lst@lefthss
\expandafter\lst@FillOutputBox\the\lst@token\@empty
\lst@righthss}%
\lst@CalcLostSpaceAndOutput}}%
\lst@ResetToken
\lst@AddToHook{OutputBox}{% init
\def\lst@gobbledwhitespacetrue{\global\let\lst@ifgobbledws\iftrue}
\def\lst@gobbledwhitespacefalse{\global\let\lst@ifgobbledws\iffalse}
\lst@AddToHookExe{InitBOL}{\lst@gobbledwhitespacefalse}% init

Delaying the output means saving the character string somewhere and pushing it back when necessary. We may also attach the string to the next output box without affecting style detection: both will be printed in the style of the upcoming output. We will call this ‘merging’.

\lst@Delay \lst@Merge

To delay or merge \#1, we process it as usual and simply save the state in macros. For delayed characters we also need the currently ‘active’ output routine. Both definitions first check whether there are already delayed or ‘merged’ characters.

\def\lst@Delay#1{%
\lst@CheckDelay
#1%
\lst@GetOutputMacro\lst@delayedoutput
\edef\lst@delayed{\the\lst@token}%
\edef\lst@delayedlength{\the\lst@length}%
\lst@ResetToken}
\def\lst@Merge#1{%
\lst@CheckMerge
#1%
\edef\lst@merged{\the\lst@token}%
\edef\lst@mergedlength{\the\lst@length}%
\lst@ResetToken}
Here we put the things together again.

We need to print delayed characters. The mode depends on the current output macro. If it equals the saved definition, we put the delayed characters in front of the character string (we merge them) since there has been no letter-to-other or other-to-letter leap. Otherwise we locally reset the current character string, merge this empty string with the delayed one, and output it.

All this is easier for \lst@merged.

It’s time to deal with fixed and flexible column modes. A couple of open definitions are now filled in.

switches to the fixed column format. The definitions here control how the output of the above definitions looks like.

Filling up a fixed mode box is easy.
While not reaching the end (\@empty from above), we insert dynamic space, output the argument and call the submacro again.

\lst@FillFixed@#1{\ifx\@empty#1\else \lst@hss#1\expandafter\lst@FillFixed@ \fi}

\lst@column@flexible The first flexible format.
\def\lst@column@flexible{\lst@flexible\true \lst@width\lst@widthflexible\relax \let\lst@OutputLostSpace\lst@UseLostSpace \let\lst@FillOutputBox\@empty \let\lst@hss\@empty \let\lst@hbox\hbox}

\lst@column@fullflexible This column format inserts no lost space except at the beginning of a line.
\def\lst@column@fullflexible{\lst@column@flexible \def\lst@OutputLostSpace{\lst@ifnewline \lst@UseLostSpace\fi}\let\lst@leftinsert\@empty \let\lst@rightinsert\@empty}

\lst@column@spaceflexible This column format only inserts lost space by stretching (invisible) existing spaces; it does not insert lost space between identifiers and other characters where the original does not have a space. It was suggested by Andrei Alexandrescu.
\def\lst@column@spaceflexible{\lst@column@flexible \def\lst@OutputLostSpace{\lst@ifwhitespace \ifx\lst@outputspace\lst@visiblespace \else \lst@UseLostSpace \fi\else \lst@ifnewline \lst@UseLostSpace\fi\fi}\let\lst@leftinsert\@empty \let\lst@rightinsert\@empty}

Thus, we have the column formats. Now we define macros to use them.
\lst@outputpos This macro sets the ‘output-box-positioning’ parameter (the old key outputpos). We test for l, c and r. The fixed formats use \lst@lefthss and \lst@righthss, whereas the flexibles need \lst@leftinsert and \lst@rightinsert.
\def\lst@outputpos@#1\relax{% \def\lst@lefthss{\ifx\@empty#1\else \lst@hss\fi}\let\lst@righthss\lst@lefthss \let\lst@rightinsert\lst@leftinsert \lst@flexible\true\relax \def\lst@OutputLostSpace{\lst@ifnewline \lst@UseLostSpace\fi}\let\lst@leftinsert\lst@InsertLostSpace \let\lst@rightinsert\lst@InsertLostSpace \let\lst@hss\@empty \let\lst@hbox\hbox\relax \fi\else %

\lst@outputpos@1\#2\relax{% \def\lst@lefthss{\lst@hss}\let\lst@rightinsert\lst@leftinsert \lst@flexible\true\relax \def\lst@OutputLostSpace{\lst@ifnewline \lst@UseLostSpace\fi}\let\lst@leftinsert\lst@InsertHalfLostSpace \let\lst@rightinsert\lst@InsertLostSpace \let\lst@hss\@empty \let\lst@hbox\hbox\relax \fi\else %

\lst@outputpos@1\#1\relax{% \let\lst@rightinsert\lst@leftinsert \lst@flexible\true\relax \def\lst@OutputLostSpace{\lst@ifnewline \lst@UseLostSpace\fi}\let\lst@leftinsert\lst@InsertHalfLostSpace \let\lst@rightinsert\lst@InsertLostSpace \let\lst@hss\@empty \let\lst@hbox\hbox\relax \fi\else %

\lst@outputpos@1\#0\relax{% \let\lst@rightinsert\lst@leftinsert

\lst@flexible\true

\lst@width\lst@widthflexible\relax

\let\lst@OutputLostSpace\lst@UseLostSpace

\let\lst@FillOutputBox\@empty

\let\lst@hss\@empty

\let\lst@hbox\hbox

\lst@column@flexible

\lst@column@fullflexible

\lst@column@spaceflexible

Thus, we have the column formats. Now we define macros to use them.
\lst@if\let\lst@lefthss\@empty\let\lst@leftinsert\@empty\ifx #1l\else \PackageWarning{Listings}{Unknown positioning for output boxes}\fi\fi\fi
\lst@ifflexible
indicates the column mode but does not distinguish between different fixed or flexible modes.
\def\lst@flexibletrue{\let\lst@ifflexible\iftrue}
\def\lst@flexiblefalse{\let\lst@ifflexible\iffalse}
columns This is done here: check optional parameter and then build the control sequence of the column format.
\lst@Key{columns}{[c]\{fixed\}}{\lstKV@OptArg[\]{#1}{%}
\ifx\@empty##1\@empty\else \lst@outputpos##1\relax\relax \fi\expandafter\let\expandafter\lst@arg\csname\@lst@column@##2\endcsname
We issue a warning or save the definition for later.
\iffalse\def\lst@arg\ifx\lst@arg\relax\PackageWarning{Listings}{Unknown column format ‘##2’}\else\lst@ifflexible\let\lst@columnsflexible\lst@arg\else\let\lst@columnsfixed\lst@arg\fi\fi
\let\lst@columnsfixed\lst@column@fixed % init
\let\lst@columnsflexible\lst@column@flexible % init
flexiblecolumns Nothing else but a key to switch between the last flexible and fixed mode.
\lst@Key{flexiblecolumns}\relax[t]{%\lstKV@SetIf{#1}\lst@ifflexible\lst@ifflexible \lst@columnsflexible\lst@arg\else \lst@columnsfixed\lst@arg\fi
\def\lst@NewLine{\ifx\lst@OutputBox\@gobble\else\par\noindent\hbox{}\fi
14.4 New lines
\lst@newlines This counter holds the number of ‘new lines’ (cr+lf) we have to perform.
\newcount\lst@newlines
\lst@AddToHook{InitVars}{\global\lst@newlines\z@}
\lst@AddToHook{InitVarsBOL}{\global\advance\lst@newlines\@ne}
\lst@NewLine This is how we start a new line: begin new paragraph and output an empty box. If low-level definition \lst@OutputBox just gobbles the box, we don’t start a new line. This is used to drop the whole output.
\def\lst@NewLine{%\ifx\lst@OutputBox\@gobble\else\par\noindent\hbox{}%\fi
109
Define \lst@newlinetrue and reset if after output.
\def\lst@newlinetrue{\global\let\lst@ifnewline\iftrue}
\lst@AddToHookExe{PostOutput}{\global\let\lst@ifnewline\iffalse}% init
\lst@TrackNewLines
If \lst@newlines is positive, we execute the hook and insert the new lines.
\def\lst@TrackNewLines{\ifnum\lst@newlines>\z@
\lsthk@OnNewLine
\lst@DoNewLines\fi}
\lst@AddToHook{OnNewLine}{}% init
emptylines
Adam Prugel-Bennett asked for such a key—if I didn’t misunderstood him. We check for the optional star and set \lst@maxempty and switch.
\lst@Key{emptylines}\maxdimen{\@ifstar{\lst@true\@tempcnta\@gobble#1\relax\lst@GobbleNil}\@tempcnta#1\relax\lst@GobbleNil}#1\@nil
\advance\@tempcnta\@one
\edef\lst@maxempty{\the\@tempcnta\relax}
\let\lst@ifpreservenumber\lst@if
\lst@DoNewLines
First we take care of \lst@maxempty and then of the remaining empty lines.
\def\lst@DoNewLines{\@whilenum\lst@newlines>\lst@maxempty \do
\lst@ifpreservenumber
\lsthk@OnEmptyLine
\global\advance\c@lstnumber\lst@advancelstnum\fi
\global\advance\lst@newlines\m@ne}
\@whilenum \lst@newlines>\@one \do\lsthk@OnEmptyLine \lst@NewLine\fi
\@whilenum \lst@newlines>\z@ \lst@NewLine \fi}
\lst@AddToHook{OnEmptyLine}{}% init

14.5 High-level output
identifierstyle
A simple key.
\lst@Key{identifierstyle}{\def\lst@identifierstyle{#1}}
\lst@AddToHook{EmptyStyle}{\let\lst@identifierstyle\@empty}
\lst@GotoTabStop
Here we look whether the line already contains printed characters. If true, we output a box with the width of a blank space.
\def\lst@GotoTabStop{\@ifnum\lst@newlines=\z@\setbox\@tempboxa\hbox{\lst@outputspace}\setbox\@tempboxa\hbox to\wd\@tempboxa{\{\lst@currstyle{\hss}\}}\lst@CalcLostSpaceAndOutput}
\def\lst@CalcLostSpaceAndOutput{It’s probably not clear why it is sufficient to output a single space to go to the next tabulator stop. Just note that the space lost by this process is ‘lost space’
in the sense above and therefore will be inserted before the next characters are output.

\else
Otherwise (no printed characters) we only need to advance \lst@lostspace, which is inserted by \lst@OutputToken above, and update the column.

\global\advance\lst@lostspace \lst@length \lst@width
\global\advance\lst@column \lst@length \z@
\fi

Note that this version works also in flexible column mode. In fact, it’s mainly the flexible version of listings 0.20.

To do: Use \lst@ifnewline instead of \ifnum\lst@newlines=\z?"?

\lst@OutputOther becomes easy with the previous definitions.
\def\lst@OutputOther{%
\lst@CheckDelay
\ifnum\lst@length=\z\else
 \let\lst@thestyle\lst@currstyle
 \lsthk@OutputOther
 \lst@OutputToken
 \fi}
\lst@AddToHook{OutputOther}{}% init
\let\lst@currstyle\relax % init
\lst@Output We might use identifier style as default.
\def\lst@Output{%
\lst@CheckDelay
\ifnum\lst@length=\z\else
 \ifx\lst@currstyle\relax
 \let\lst@thestyle\lst@identifierstyle
 \else
 \let\lst@thestyle\lst@currstyle
 \fi
 \lsthk@Output
 \lst@OutputToken
 \fi
 \let\lst@lastother\relax}
\lst@AddToHook{Output}{}% init
\lst@GetOutputMacro Just saves the output macro to be used.
\def\lst@GetOutputMacro#1{%
\lst@ifletter \global\let#1\lst@Output\else \global\let#1\lst@OutputOther\fi}
\lst@PrintToken outputs the current character string in letter or nonletter mode.
\def\lst@PrintToken{%
\lst@ifletter \lst@Output \lst@letterfalse
 \else \lst@OutputOther \let\lst@lastother@empty \fi}
\lst@XPrintToken is a special definition to print also merged characters.

\begin{verbatim}
\def\lst@XPrintToken{\lst@PrintToken \lst@CheckMerge
  \ifnum\lst@length=\z@else \lst@PrintToken \fi}
\end{verbatim}

\subsection*{14.6 Dropping the whole output}
\lst@BeginDropOutput It’s sometimes useful to process a part of a listing as usual, but to drop the output. This macro does the main work and gets one argument, namely the internal mode it enters. We save \lst@newlines, restore it \aftergroup and redefine one macro, namely \lst@OutputBox. After a bug report from Gunther Schmidl
\begin{verbatim}
\def\lst@BeginDropOutput#1{\xdef\lst@BDOnewlines{\the\lst@newlines}\
  \global\let\lst@BDOifnewline\lst@ifnewline\lst@EnterMode{#1}{\lst@modetrue
    \let\lst@OutputBox\@gobble\aftergroup\lst@BDORestore}}
\end{verbatim}

Restoring the date is quite easy:
\begin{verbatim}
\def\lst@BDORestore{\global\lst@newlines\lst@BDOnewlines\global\let\lst@ifnewline\lst@BDOifnewline}
\end{verbatim}
\lst@EndDropOutput is equivalent to \lst@LeaveMode.
\begin{verbatim}
\let\lst@EndDropOutput\lst@LeaveMode
\end{verbatim}

\subsection*{14.7 Writing to an external file}
\begin{verbatim}
\def\lst@BeginAspect{writefile}
\end{verbatim}
\begin{verbatim}
\lst@WF \lst@WFtoken The contents of the token will be written to file.
\end{verbatim}
\begin{verbatim}
\neutronks\lst@WFtoken % global\lst@AddToHook{InitVarsBOL}{\global\lst@WFtoken{}}\newwrite\lst@WF\global\let\lst@WFifopen\iffalse % init
\end{verbatim}
\begin{verbatim}
\lst@WFWriteToFile To do this, we have to expand the contents and then expand this via \edef. Empty \lst@OM ensures that special characters (underscore, dollar, etc.) are written correctly.
\gdef\lst@WFWriteToFile{\begingroup\let\lst@UM\@empty\expandafter\edef\expandafter\lst@temp\expandafter{\the\lst@WFtoken}\immediate\write\lst@WF{\lst@temp}\endgroup\global\lst@WFtoken{}}
\end{verbatim}
\lst@WFAppend

Similar to \lst@Append but uses \lst@WFtoken.

\lst@BeginWriteFile \lst@BeginAlsoWriteFile

use different macros for \lst@OutputBox (not) to drop the output.

\lst@WFBegin

Here ... 

\begin{group} \catcode'\^^I=11 \end{group}

\gdef\lst@WFBegin#1#2{% 
\begingroup \let\lst@OutputBox#1% ... we have to update \lst@WFtoken and ...
\def\lst@Append##1{% 
\advance\lst@length\@ne \expandafter\lst@token\expandafter{\the\lst@token##1}% 
\ifx ##1\lst@outputspace \else \lst@WFAppend##1% \fi} 
\lst@lAddTo\lst@PreGotoTabStop{\lst@WFAppend{^^I}} 
\lst@lAddTo\lst@ProcessSpace{\lst@WFAppend{ }} . . . need different ‘EOL’ and ‘DeInit’ definitions to write the token register to file.
\let\lst@DeInit\lst@WFDeInit \let\lst@MProcessListing\lst@WFMProcessListing

Finally we open the file if necessary.
\lst@WFifopen\else
\immediate\openout\lst@WF=#2\relax \global\let\lst@WFifopen\iftrue \@gobbletwo\fi\fi
\fi}
\endgroup

\lst@EndWriteFile

closes the file and restores original definitions.
\lst@WFDeInit

\begin{group}
\immediate\closeout\lst@WF \endgroup
\global\let\lst@WFifopen\iffalse

\lst@WFMProcessListing

write additionally \lst@WFtoken to external file.
\lst@WFDeInit

\begin{group}
\global\let\lst@WFMProcessListing\lst@WFMProcessListing
\lst@AddToAtTop\lst@WFMProcessListing{\lst@WFWriteToFile}
\lst@AddToAtTop\lst@WFDeInit{\ifnum\lst@length=\z@\else \lst@WFWriteToFile \fi}
\endgroup

\lst@EndAspect

\langle /misc \rangle

\section{Character classes}

In this section, we define how the basic character classes do behave, before turning over to the selection of character tables and how to specialize characters.
15.1 Letters, digits and others

\lst@ProcessLetter We put the letter, which is not a whitespace, into the output queue.
\def\lst@ProcessLetter{\lst@whitespacefalse \lst@AppendLetter}

\lst@ProcessOther Ditto.
\def\lst@ProcessOther{\lst@whitespacefalse \lst@AppendOther}

\lst@ProcessDigit A digit appends the character to the current character string. But we must use the right macro. This allows digits to be part of an identifier or a numerical constant.
\def\lst@ProcessDigit{\lst@whitespacefalse \lst@ifletter \expandafter\lst@AppendLetter \else \expandafter\lst@AppendOther\fi}

\lst@ifwhitespace indicates whether the last processed character has been white space.
\def\lst@whitespacetrue{\global\let\lst@ifwhitespace\iftrue}
\def\lst@whitespacefalse{\global\let\lst@ifwhitespace\iffalse}
\lst@AddToHook{InitVarsBOL}{\lst@whitespacetrue}

15.2 Whitespaces

Here we have to take care of two things: dropping empty lines at the end of a listing and the different column formats. Both use \lst@lostspace. Lines containing only tabulators and spaces should be viewed as empty. In order to achieve this, tabulators and spaces at the beginning of a line don’t output any characters but advance \lst@lostspace. Whenever this dimension is positive we insert that space before the character string is output. Thus, if there are only tabulators and spaces, the line is ‘empty’ since we haven’t done any output.

We have to do more for flexible columns. Whitespaces can fix the column alignment: if the real line is wider than expected, a tabulator is at least one space wide; all remaining space fixes the alignment. If there are two or more space characters, at least one is printed; the others fix the column alignment.

Tabulators are processed in three stages. You have already seen the last stage \lst@GotoTabStop. The other two calculate the necessary width and take care of visible tabulators and spaces.

tabsize We check for a legal argument before saving it. Default tabsize is 8 as proposed by Rolf Niepraschck.
\def\lst@tabsize{8}
\lst@Key{tabsize}{8} \{\def\lst@tabsize{#1}\else
\PackageError{Listings}{Strict positive integer expected}{You can’t use ‘#1’ as tabsize. \@ehc} \fi}

showtabs Two more user keys for tab control.
tab \lst@Key{showtabs}f[t]{\lstKV@SetIf{#1}\lst@ifshowtabs
\kern.06em\hbox{\vrule\@height.3ex}\hrulefill\hbox{\vrule\@height.3ex}}
\def\lst@tab{#1}
A tabulator outputs the preceding characters, which decrements \texttt{\lst@pos} by the number of printed characters.

Then we calculate how many columns we need to reach the next tabulator stop: we add \texttt{\lst@tabsize} until \texttt{\lst@pos} is strictly positive. In other words, \texttt{\lst@pos} is the column modulo \texttt{\lst@tabsize} and we’re looking for a positive representative. We assign it to \texttt{\lst@length} and reset \texttt{\lst@pos} in the submacro.

Visible tabs print \texttt{\lst@tab}.

Spaces are implemented as described at the beginning of this subsection. But first we define some user keys.

The first macro is a default definition, ...
We look whether spaces fix the column alignment or not. In the latter case we append a space; otherwise ... Andrei Alexandrescu tested the spaceflexible column setting and found a bug that resulted from \lst@PrintToken and \lst@whitespacetrue being out of order here.

```latex
\lst@ProcessSpace
\def\lst@ProcessSpace{\%
\lst@ifkeepspaces
\lst@PrintToken
\lst@whitespacetrue
\lst@AppendOther\lst@outputspace
\lst@PrintToken
\else \ifnum\lst@newlines=\z@
... we append a ‘special space’ if the line isn’t empty.
\lst@AppendSpecialSpace
\else \ifnum\lst@length=\z@
If the line is empty, we check whether there are characters in the output queue. If there are no characters we just advance \lst@lostspace. Otherwise we append the space.
\global\advance\lst@lostspace\lst@width
\global\advance\lst@pos\m@ne
\lst@whitespacetrue
\else
\lst@AppendSpecialSpace
\fi \fi}
```

Note that this version works for fixed and flexible column output.

```latex
\lst@AppendSpecialSpace
\def\lst@AppendSpecialSpace{\%
\lst@ifwhitespace
\lst@PrintToken
\global\advance\lst@lostspace\lst@width
\global\advance\lst@pos\m@ne
\lst@gobbledwhitespacetrue
\else
\lst@PrintToken
\lst@whitespacetrue
\lst@AppendOther\lst@outputspace
\lst@PrintToken
\fi}
```

If there are at least two white spaces, we output preceding characters and advance \lst@lostspace to avoid alignment problems. Otherwise we append a space to the current character string. Also, \lst@whitespacetrue has been moved after \lst@PrintToken so that the token-printer can correctly check whether it is printing whitespace or not; this was preventing the spaceflexible column setting from working correctly.

```latex
\lst@Key{formfeed}{\bigbreak}{\def\lst@formfeed{#1}}
```

Form feeds has been introduced after communication with Jan Braun.

```latex
\lst@Key{formfeed}{\bigbreak}{\def\lst@formfeed{#1}}
```
Here we execute some macros according to whether a new line has already begun or not. No \lst@EOLUpdate is used in the else branch anymore—Kalle Tuulos sent the bug report.

\def\lst@ProcessFormFeed{\ifnum\lst@newlines=1\lst@EOLUpdate \lsthk@InitVarsBOL\fi\lst@formfeed\lst@whitespacetrue}

\section{Character tables}
\subsection{The standard table}

The standard character table is selected by \lst@SelectStdCharTable, which expands to a token sequence ...\def A{\lst@ProcessLetter A}... where the first A is active and the second has catcode 12. We use the following macros to build the character table.

\lst@CCPut⟨class macro⟩⟨c₁⟩...⟨cₖ⟩\z@

extends the standard character table by the characters with codes ⟨c₁⟩...⟨cₖ⟩ making each character use ⟨class macro⟩. All these characters must be printable via \char⟨ci⟩.

\lst@CCPutMacro⟨class⟩⟨c_i⟩⟨definition⟩...@empty@empty

also extends the standard character table: the character ⟨cᵢ⟩ will use ⟨classᵢ⟩ and is printed via ⟨definitionᵢ⟩. These definitions must be ⟨spec. token⟩s in the sense of section 9.5.

\def\lst@Def#1{\lccode'~=#1\lowercase{\def~}}
\def\lst@Let#1{\lccode'~=#1\lowercase{\let~}}

The definition of the space below doesn’t hurt anything. But other aspects, for example lineshape and formats, redefine also the macro \space. Now, if LATEX calls \@try@load@fontshape, the .log messages would show some strange things since LATEX uses \space in these messages. The following addition ensures that \space expands to a space and not to something different. This was one more bug reported by Denis Girou.

\lst@AddToAtTop{\@try@load@fontshape}{\def\space{\ }}

\lst@SelectStdCharTable

The first three standard characters. \lst@Let has been replaced by \lst@Def after a bug report from Chris Edwards.

\lst@CCPut

The first argument gives the character class, then follow the codes.

Joseph Wright pointed to a bug which came up on TeX StackExchange (http://tex.stackexchange.com/questions/302437/textcase-listings-and-tilde).
Other than in \lst@CCPutMacro the \lccode settings weren’t local and caused the error.

\begin{verbatim}
985 \def\lst@CCPut#1#2{\% 
986 \ifnum#2=\z@ \else 
987 \begingroup\lccode'\~=#2\lccode'\/=\endgroup\lst@CCPut@~{#1/}\%
988 \fi 
989 \def\lst@CCPut@#1#2{\lst@lAddTo\lst@SelectStdCharTable{\def#1{#2}}}
990 \end{verbatim}

Now we insert more standard characters.

\begin{verbatim}
993 \lst@CCPut \lst@ProcessOther \{21\} \{22\} \{28\} \{29\} \{2B\} \{2E\} \{2F\}
994 \{3A\} \{3B\} \{3D\} \{3F\} \{5B\} \{5D\}
995 \z@
996 \lst@CCPut \lst@ProcessDigit \{30\} \{31\} \{32\} \{33\} \{35\} \{36\} \{37\} \{38\} \{39\}
997 \z@
998 \lst@CCPut \lst@ProcessLetter \{40\} \{41\} \{42\} \{43\} \{44\} \{45\} \{46\} \{47\}
999 \{48\} \{49\} \{4A\} \{4B\} \{4C\} \{4D\} \{4E\} \{4F\}
1000 \{50\} \{51\} \{52\} \{53\} \{54\} \{55\} \{56\} \{57\}
1001 \{58\} \{59\} \{5A\}
1002 \{61\} \{62\} \{63\} \{64\} \{65\} \{66\} \{67\}
1003 \{68\} \{69\} \{6A\} \{6B\} \{6C\} \{6D\} \{6E\} \{6F\}
1004 \{70\} \{71\} \{72\} \{73\} \{74\} \{75\} \{76\} \{77\}
1005 \{78\} \{79\} \{7A\}
1006 \z@
\end{verbatim}

Now we come to a delicate point. The characters not inserted yet aren’t printable (\_, \$, \ldots) or aren’t printed well (\*, \-, \ldots) if we enter these characters. Thus we use proper macros to print the characters. Works perfectly. The problem is that the current character string is printable for speed, for example \_ is already replaced by a macro version, but the new keyword tests need the original characters.

The solution: We define \def _\{\lst@ProcessLetter\lst@um_\} where the first underscore is active and the second belongs to the control sequence. Moreover we have \def \lst@um_\{\lst@UM_\} where the second underscore has the usual meaning. Now the keyword tests can access the original character simply by making \lst@UM empty. The default definition gets the following token and builds the control sequence \lst@um_\, which we’ll define to print the character. Easy, isn’t it?

The following definition does all this for us. The first parameter gives the character class, the second the character code, and the last the definition which actually prints the character. We build the names \lst@um_ and \lst@um_\, and give them to a submacro.

\begin{verbatim}
1010 \def\lst@CCPutMacro#1#2#3\{
1011 \ifnum#2=\z@ \else 
1012 \begingroup\lccode'\~=#2\relax \lccode'\/=\endgroup\lst@CCPutMacro@ \% 
1013 \lowercase\endgroup\expandafter\lst@CCPutMacro\csname@lst\um\expandafter\endcsname\%
1014 \csname@lst\um@\expandafter\endcsname /\#1\#3\%
1015 \expandafter\lst@CCPutMacro\end{verbatim}

\lst@CCPutMacro
The arguments are now \lst@um_, \lst@um_@, nonactive character, active character, character class and printing definition. We add \def_\lst@ProcessLetter \lst@um_\@ to \lst@SelectStdCharTable (and similarly other special characters), define \def\lst@um_{\ lst@UM_{\ lst@UM_\@}} and \lst@um_0. \def\lst@CCPutMacro@#1#2#3#4#5#6{% \lst@lAddTo\lst@SelectStdCharTable{\def#4{#5#1}}% \def#1{\lst@UM#3}% \def#2{#6}} The default definition of \lst@UM: \def\lst@UM#1{\csname\@lst@um#1@\endcsname} And all remaining standard characters. \lst@CCPutMacro \lst@ProcessOther{"23}# \lst@ProcessLetter{"24}\textdollar \lst@ProcessOther{"25}\% \lst@ProcessOther{"26}& \lst@ProcessOther{"27}{\lst@ifupquote \textquotesingle \else \char39\relax \fi} \lst@ProcessOther{"2A}{\lst@ttfamily\*\textasteriskcentered} \lst@ProcessOther{"2D}{\lst@ttfamily{-}{\textbackslash}} \lst@ProcessOther{"3C}{\lst@ttfamily<\textless} \lst@ProcessOther{"3E}{\lst@ttfamily>\textgreater} \lst@ProcessOther{"5C}{\lst@ttfamily\char92\textbackslash} \lst@ProcessLetter{"5E} \lst@ProcessOther{"5F}{\lst@ttfamily\char95\textunderscore} \lst@ProcessOther{"60}{\lst@ifupquote \textasciigrave \else \char96\relax \fi} \lst@ProcessOther{"7B}{\lst@ttfamily\char123\textbraceleft} \lst@ProcessOther{"7C}{\lst@ttfamily\char125\textbraceright} \lst@ProcessOther{"7D}{\lst@ttfamily\char126\textbar} \lst@ProcessOther{"7E}{\lst@ttfamily\textasciitilde} \lst@ProcessOther{"7F}-- \@empty\z@\@empty The default definition of \lst@UM:\def\lst@UM#1{\csname\@lst@um#1@\endcsname} \lst@Key{upquote}{false}[t]{\lstKV@SetIf{#1}\lst@ifupquote \lst@ifupquote \@ifundefined{textasciigrave}{}{\let\KV@lst@upquote\@gobble} \lst@AddToHook{Init}{\edef\ttdefault{\ttdefault}} \@empty\z@\@empty What is this ominous macro? It prints either the first or the second argument. In rttfamily it ensures that ---- is typeset ---- and not ---- as in version 0.17. Bug encountered by Dr. Jobst Hoffmann. Furthermore I added \relax after receiving an error report from Magnus Lewis-Smith. \def\lst@ttfamily#1#2{%\ifx\f@family\ttdefault#1\relax\else#2\fi} \ttdefault is defined \long, so the \ifx doesn’t work since \f@family isn’t \long! We go around this problem by redefining \ttdefault locally: \lst@AddToHook{Init}{\edef\ttdefault{\ttdefault{\ttdefault}}} \lst@Key{upquote}{false}[t]{\lstKV@SetIf{#1}\lst@ifupquote \lst@ifupquote \@ifundefined{textasciigrave}{}{\let\KV@lst@upquote\@gobble} upquote is used above to decide which quote to print. We print an error message if the necessary textcomp commands are not available. This key has been added after an email from Frank Mittelbach.
If an `upquote` package is loaded, the `upquote` option is enabled by default.

\AtBeginDocument{%
\@ifpackageloaded{upquote}{\RequirePackage{textcomp}\
\lstset{upquote}}{}%
\@ifpackageloaded{upquote2}{\lstset{upquote}}{}}

\lst@ifactivechars
\def\lst@activecharstrue{\let\lst@ifactivechars\iftrue}
\def\lst@activecharsfalse{\let\lst@ifactivechars\iffalse}
\lst@activecharstrue
\lst@SelectCharTable
We select the standard character table and switch to active catcodes.
\def\lst@SelectCharTable{\lst@SelectStdCharTable\
\lst@ifactivechars
\catcode9\active \catcode12\active \catcode13\active
\@tempcnta=32\relax
\@whilenum\@tempcnta<128\do
{\catcode\@tempcnta\active\advance\@tempcnta\@ne}\
\fi
\lst@ifec \lst@DefEC \fi
The following line and the according macros below have been added after a bug report from Frédéric Boulanger. The assignment to \do@noligs was changed to \do after a bug report from Peter Ruckdeschel. This bugfix was kindly provided by Timothy Van Zandt.
\let\do\lst@do@noligs \verbatim@nolig@list
There are two ways to adjust the standard table: inside the hook or with \lst@DeveloperSCT. We use these macros and initialize the backslash if necessary. \lst@DefRange has been moved outside the hook after a bug report by Michael Bachmann.
\lst@SelectCharTable
\lst@AddToHook{SetLanguage}{\let\lst@DeveloperSCT\@empty}
\lst@do@noligs
To prevent ligatures, this macro inserts the token \lst@NoLig in front of \lst@Process ⟨whatever⟩⟨spec. token⟩. This is done by \verbatim@nolig@list
\lst@NoLig When this extra macro is processed, it adds \lst@nolig to the output queue without increasing its length. For keyword detection this must expand to nothing if \lst@UM is empty.

\lst@SaveOutputDef To get the (spec. token) meaning of character #1, we look \lst@SelectStdCharTable, get the replacement text, strip off the character class via \@gobble, and assign the meaning. Note that you get a “runaway argument” error if an illegal (character code)=#1 is used.

\lst@SaveOutputDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\def\lst@temp##1\def~##2##3\relax}{\global\expandafter\let\expandafter#2\@gobble##2\relax}:\expandafter\lst@temp\lst@SelectStdCharTable\relax}

\lstum@backslash A commonly used character.

\lstum@backslash

15.3.2 National characters

\lst@Key{extendedchars}{true}[t]{\lstKV@SetIf{#1}\lst@ifec}

\lst@DefEC Currently each character in the range 128–255 is treated as a letter.
Reaching end of list (`^00`) we terminate the loop. Otherwise we do the same as in `\lst@CCTPut` if the character is not active. But if the character is active, we save the meaning before redefinition.

\begin{verbatim}
\def\lst@CCECUse#1#2{\ifnum'#2=\z@\expandafter\@gobbletwo\else\ifnum\catcode'#2=\active\lccode'\~='#2\lccode'\~/=#2\lowercase{\lst@CCECUse@#1~/}\else\lst@ifactivechars \catcode'#2=\active \fi\lccode'\~='#2\lccode'\~/=#2\lowercase{\def~{#1/}}\fi\fi\lst@CCECUse#1}
\end{verbatim}

We save the meaning as mentioned. Here we must also use the `\lst@UM construction` since extended characters could often appear in words = identifiers. Bug reported by Denis Girou.

\def\lst@CCECUse@#1#2#3{\expandafter\def\csname\@lst@EC#3\endcsname{\lst@UM#3}\expandafter\let\csname\@lst@um#3@\endcsname#2\edef#2{\noexpand#1\expandafter\noexpand\csname\@lst@EC#3\endcsname}}

Daniel Gerigk and Heiko Oberdiek reported an error and a solution, respectively.

\section{Catcode problems}

Anders Edenbrandt found a bug with `.fd`-files. Since we change catcodes and these files are read on demand, we must reset the catcodes before the files are input. We use a local redefinition of `\nfss@catcodes`.

\begin{verbatim}
\def\lst@nfss@catcodes{\lst@makeletter
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz\relax\@makeother (\@makeother )\@makeother ,\@makeother :\@makeother \&\@makeother 0\@makeother 1\@makeother 2\@makeother 3\@makeother 4\@makeother 5\@makeother 6\@makeother 7\@makeother 8\@makeother 9\@makeother =\lst@nfss@catcodes}\relax\@makeother \&\@makeother =\lst@makeletter
\end{verbatim}

The investigation of a bug reported by Christian Gudrian showed that the equal sign needs to have ‘other’ catcode, as assigned above. Svend Tollak Munkejord reported problems with Lucida `.fd-files`, while Heiko Oberdiek analysed the bug, which above led to the line starting with `\@makeother (`. The name of `\lst@makeletter` is an imitation of L\TeX’s `\makeother`.

Another problem was first reported by Marcin Kasperski. It is also catcode related and Donald Arseneau let me understand it. The point is that \TeX seems to use the
currently active catcode table when it writes non-\immediate \write to file and not the catcodes involved when reading the characters. So a section heading \La was written \La if a listing was split on two pages since a non-standard catcode table was in use when writing \La to file, the previously attached catcodes do not matter. One more bug was that accents in page headings or footers were lost when a listing was split on two pages. Denis Girou found this latter bug. A similar problem with the tilde was reported by Thorsten Vitt.

We can choose between three possibilities. Donald Arseneau noted a bug here in the \ifcase argument.

\begin{verbatim}
\lst@Key{useoutput}{2}{\edef\lst@useoutput{\ifcase0#1 0\or 1\else 2\fi}}
\end{verbatim}

The first does not modify the existing output routine.

\begin{verbatim}
\lst@AddToHook{Init}
\lst@ifcase\lst@useoutput{\relax
\or
\end{verbatim}

The second possibility is as follows: We interrupt the current modes—in particular \lst@Pmode with modified catcode table—, call the original output routine and reenter the mode. This must be done with a little care. First we have to close the group which \TeX opens at the beginning of the output routine. A single \egroup gives an ‘unbalanced output routine’ error. But \expandafter\egroup works. Again it was Donald Arseneau who gave the explanation: The \expandafter set the token type of \bgroup to \texttt{backed_up}, which prevents \TeX’s from recovering from an unbalanced output routine. Heiko Oberdiek reported that \texttt{csname egroup endcsname} does the trick, too.

However, since \TeX checks the contents of \texttt{box 255} when we close the group (‘output routine didn’t use all of \texttt{box 255}’), we have to save it temporarily.

\begin{verbatim}
\output{\global\setbox\lst@gtempboxa\box\@cclv
\expandafter\egroup
\aftergroup\pagegoal\aftergroup\vsize
\aftergroup\lst@ReenterModes\aftergroup\lst@RestoreToken}%
\else
\end{verbatim}

The third option is to restore all catcodes and meanings inside a modified output routine and to call the original routine afterwards.

\begin{verbatim}
\output{\lst@RestoreOrigCatcodes\lst@ifec \lst@RestoreOrigExtendedCatcodes \fi
\lst@ifec\lst@RestoreToken\fi}
\end{verbatim}
Note that this output routine isn’t used too often. It is executed only if it’s possible that a listing is split on two pages: if a listing ends at the bottom or begins at the top of a page, or if a listing is really split.

```latex
\def\lst@GetChars#1#2#3{% \let#1\@empty \@tempcnta#2 \relax \@tempcntb#3 \relax \loop \ifnum\@tempcnta<\@tempcntb \relax \lst@lExtend#1{\expandafter\catcode\the\@tempcnta=} \lst@lExtend#1{\the\catcode\@tempcnta\relax} \ifnum\the\catcode\@tempcnta=\active \begingroup\lccode'~\=\@tempcnta \lowercase\endgroup \lst@lExtend#1{\expandafter\let\expandafter~\csname lstecs@\the\@tempcnta\endcsname} \expandafter\let\csname lstecs@\the\@tempcnta\endcsname~} \advance\@tempcnta\@ne \repeat}
```

As per a bug report by Benjamin Lings, we deactivate \outer definition of ^^^L temporarily (inside and outside of \lst@ScanChars) and restore the catcode at end of package via the \lst@RestoreCatcodes command.

```latex
\begingroup \catcode12=\active \let^^L\@empty \gdef\lst@ScanChars{% \let\lsts@ssL^^L \def^^L{\par} \lst@GetChars\lst@RestoreOrigCatcodes\@ne 128 \let^^L\lsts@ssL \lst@GetChars\lst@RestoreOrigExtendedCatcodes{128}{256} \endgroup
```

The scan can be issued by hand and at the beginning of a document.

```latex
\lst@Key{rescanchars}\relax{\lst@ScanChars} \AtBeginDocument{\lst@ScanChars}
```

### 15.3.4 Adjusting the table

We begin with modifiers for the basic character classes.

```latex
\def\lst@alsoletter\relax{% \lst@DoAlso{#1}\lst@alsoletter\lst@ProcessLetter}
\def\lst@alsodigit\relax{% \lst@DoAlso{#1}\lst@alsodigit\lst@ProcessDigit}
\def\lst@alsoother\relax{% \lst@DoAlso{#1}\lst@alsoother\lst@ProcessOther}
```

This is done at SelectCharTable and every language selection the macros get empty.

```latex
\lst@AddToHook{SelectCharTable}
```

124
The service macro starts a loop and ...
\texttt{\def\lst@DoAlso#1#2#3{\lst@DefOther\lst@arg{#1}\let#2\@empty\expandafter\lst@DoAlso@\expandafter#2\expandafter#3\lst@arg\relax}}
\texttt{\def\lst@DoAlso@#1#2#3{\ifx\relax#3\expandafter\@gobblethree \else \ldots while not reaching \relax we use the \TeX\ technique from \\lst@SaveOutputDef to replace the class by #2. Eventually we append the new definition to #1. \begingroup \lccode'~='#3\relax \lowercase{\endgroup
\def\lst@temp##1\def~##2##3\relax{\edef\lst@arg{\def\noexpand~{\noexpand#2\expandafter
\noexpand\@gobble##2}}}}}\expandafter\lst@temp\lst@SelectStdCharTable\relax \lst@lExtend#1{\lst@arg}\fi \lst@DoAlso@}}

\lst@SaveDef\lst@DefSaveDef\lst@LetSaveDef These macros can be used in language definitions to make special changes. They save the definition and define or assign a new one. \texttt{\def\lst@SaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~}}}\texttt{\def\lst@DefSaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~\def~}}}\texttt{\def\lst@LetSaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~\let~}}}

Now we get to the more powerful definitions.
\texttt{\def\lst@CDef#1{\lst@CDef@#1}}\texttt{\def\lst@CDef@#1#2#3#4{\lst@CDefIt#1{#2}{#3}{#4#2#3}#4}}\texttt{\lst@CDefX} drops the input string. \texttt{\def\lst@CDefX#1{\lst@CDefX@#1#2#3}}\texttt{\lst@CDefIt} is the main working procedure for the previous macros. It redefines the sequence \#1#2#3 of characters. At least \#1 must be active; the other two arguments might be empty, not equivalent to empty! \texttt{\def\lst@CDefIt#1#2#3#4#5#6#7\empty{\ifx\empty#2\empty \def#1{#6\def\lst@next{#7\lst@next}}\else \ifx\empty#3\empty \expandafter\lst@DoAlso@\expandafter\@gobblethree \else ... while not reaching \relax we use the \TeX\ technique from \\lst@SaveOutputDef to replace the class by #2. Eventually we append the new definition to #1. \begingroup \lccode'~='#3\relax \lowercase{\endgroup
\def\lst@temp##1\def~##2##3\relax{\edef\lst@arg{\def\noexpand~{\noexpand#2\expandafter\noexpand\@gobble##2}}}}}\expandafter\lst@temp\lst@SelectStdCharTable\relax \lst@lExtend#1{\lst@arg}\fi \lst@DoAlso@}}

\texttt{\def\lst@SaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~}}}\texttt{\def\lst@DefSaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~\def~}}}\texttt{\def\lst@LetSaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~\let~}}}

Now we get to the more powerful definitions.
\texttt{\def\lst@CDef#1{\lst@CDef@#1}}\texttt{\def\lst@CDef@#1#2#3#4{\lst@CDefIt#1{#2}{#3}{#4#2#3}#4}}\texttt{\lst@CDefX} drops the input string. \texttt{\def\lst@CDefX#1{\lst@CDefX@#1#2#3}}\texttt{\lst@CDefIt} is the main working procedure for the previous macros. It redefines the sequence \#1#2#3 of characters. At least \#1 must be active; the other two arguments might be empty, not equivalent to empty! \texttt{\def\lst@CDefIt#1#2#3#4#5#6#7\empty{\ifx\empty#2\empty \def#1{#6\def\lst@next{#7\lst@next}}\else \ifx\empty#3\empty \expandafter\lst@DoAlso@\expandafter\@gobblethree \else ... while not reaching \relax we use the \TeX\ technique from \\lst@SaveOutputDef to replace the class by #2. Eventually we append the new definition to #1. \begingroup \lccode'~='#3\relax \lowercase{\endgroup
\def\lst@temp##1\def~##2##3\relax{\edef\lst@arg{\def\noexpand~{\noexpand#2\expandafter\noexpand\@gobble##2}}}}}\expandafter\lst@temp\lst@SelectStdCharTable\relax \lst@lExtend#1{\lst@arg}\fi \lst@DoAlso@}}

\texttt{\def\lst@SaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~}}}\texttt{\def\lst@DefSaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~\def~}}}\texttt{\def\lst@LetSaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~\let~}}}

Now we get to the more powerful definitions.
\texttt{\def\lst@CDef#1{\lst@CDef@#1}}\texttt{\def\lst@CDef@#1#2#3#4{\lst@CDefIt#1{#2}{#3}{#4#2#3}#4}}\texttt{\lst@CDefX} drops the input string. \texttt{\def\lst@CDefX#1{\lst@CDefX@#1#2#3}}\texttt{\lst@CDefIt} is the main working procedure for the previous macros. It redefines the sequence \#1#2#3 of characters. At least \#1 must be active; the other two arguments might be empty, not equivalent to empty! \texttt{\def\lst@CDefIt#1#2#3#4#5#6#7\empty{\ifx\empty#2\empty \def#1{#6\def\lst@next{#7\lst@next}}\else \ifx\empty#3\empty \expandafter\lst@DoAlso@\expandafter\@gobblethree \else ... while not reaching \relax we use the \TeX\ technique from \\lst@SaveOutputDef to replace the class by #2. Eventually we append the new definition to #1. \begingroup \lccode'~='#3\relax \lowercase{\endgroup
\def\lst@temp##1\def~##2##3\relax{\edef\lst@arg{\def\noexpand~{\noexpand#2\expandafter\noexpand\@gobble##2}}}}}\expandafter\lst@temp\lst@SelectStdCharTable\relax \lst@lExtend#1{\lst@arg}\fi \lst@DoAlso@}}

\texttt{\def\lst@SaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~}}}\texttt{\def\lst@DefSaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~\def~}}}\texttt{\def\lst@LetSaveDef#1#2{\begingroup \lccode'~=#1\relax \lowercase{\endgroup\let#2~\let~}}}

Now we get to the more powerful definitions.
For a two character sequence we test whether \langle \text{pre} \rangle and \langle \text{post} \rangle must be executed.

\def#1##1{\protect\ifx##1#2\def\lst@next{#7#4#8}\else \def\lst@next{#5##1}\fi \lst@next}\
\else

We do the same for an arbitrary character sequence—except that we have to use \lst@ifNextCharsArg instead of \ifx...\fi.

\def#1{\protect\lst@IfNextCharsArg{#2#3}{#7#4#8}{\expandafter#5\lst@eaten}}\
\fi}\
\lst@CArgX We make \#1\#2 active and call \lst@CArg.

\def\lst@CArgX#1#2\relax{\lst@DefActive\lst@arg{#1#2}\expandafter\lst@CArg\lst@arg\relax}
\lst@CArg arranges the first two arguments for \lst@CDef. We get an undefined macro and use \@empty\@empty\relax as delimiter for the submacro.

\def\lst@CArg#1#2\relax{\let#1#2}\ifx\@empty#3\@empty\def\lst@next{#6{#2{}{}}}\else \def\lst@next{#6{#2#3{#4}}}\fi \lst@next #1

\lst@CArgEmpty executes' an \@empty-delimited argument. We will use it for the delimiters.

\def\lst@CArgEmpty#1\@empty{#1}

\subsection*{15.4 Delimiters}

Here we start with general definitions common to all delimiters.

\lst@ifex\langle \text{whatever} \rangle to be true. Such switches are set false in the \ExcludeDelims hook and are handled by the individual delimiters.
\lst@DelimPrint  And this macro might help in doing so. #1 is \lst@ifex(whatever) (plus \else) or just \iffalse, and #2 will be the delimiter. The temporary mode change ensures that the characters can’t end the current delimiter or start a new one.

\lst@DelimOpen  We print preceding characters and the delimiter, enter the appropriate mode, print the delimiter again, and execute #3. In fact, the arguments #1 and #2 will ensure that the delimiter is printed only once.

\lst@DelimClose  is the same in reverse order.

\lst@BeginDelim  These definitions are applications of \lst@DelimOpen and \ lst@DelimClose: the delimiters have the same style as the delimited text.

\lst@BeginIDelim  Another application: no delimiter is printed.

\lst@DefDelims  This macro defines all delimiters and is therefore reset every language selection.

\lst@Delim  First we set default values: no \lst@modetrue, cumulative style, and no argument to \lst@Delim[DM][@](type).
The type argument is saved for later use. We check against the optional \langle style \rangle argument using #1 as default, define \lst@delimstyle and look for the optional \langle type option \rangle, which is just saved in \lst@arg.

\lst@Delim@#1[#2]{{\lst@delimtype{#2}}}\lst@true\lst@Delim@{#1}
\def\lst@Delim@#1[#2]{\gdef\lst@delimtype{#2}}\@ifnextchar[\lst@Delim@sty{\lst@Delim@sty[#1]}
\def\lst@Delim@sty[#1]{\def\lst@delimstyle{#1}\ifx\@empty#1\@empty\else\lst@Delim@sty@ #1\@nil\fi}\@ifnextchar[\lst@Delim@option\lst@Delim@delim}
\def\lst@Delim@option[#1]{\def\lst@arg{[#1]}\lst@Delim@delim}
\lst@Delim@delim

[ and ] in the replacement text above have been added after a bug report by Stephen Reindl.

The definition of \lst@delimstyle depends on whether the first token is a control sequence. Here we possibly build \lst@(style).
\def\lst@Delim@sty@#1#2\@nil{\if\relax
\else\edef\lst@delimstyle{\expandafter\noexpand\csname\@lst @\lst@delimstyle\endcsname} \fi}

\let\lst@delim\@empty\expandafter\lst@IfOneOf\lst@delimtype\relax#3\@ifundefined{\@lst @#2DM@\lst@delimtype}{{\lst@lExtend\lst@delim{\csname\@lst @#2\@\lst@delimtype \expandafter\endcsname\lst@arg}}}{{\lst@lExtend\lst@delim{\csname\@lst @#2DM@\lst@delimtype \expandafter\endcsname\lst@arg}}}\@ifnextchar[\lst@Delim@delall\lst@Delim@delim}

\lst@Delim@delim

Eventually this macro is called. First we might need to delete a bunch of delimiters.

If there is no delimiter, we might delete a subclass.
\def\lst@Delim@delim1\relax#2#3#4#5#6#7#8\{\@ifnextchar[\lst@Delim@delall\lst@Delim@delim\#1\#2\#3\#4\#5\#6\#7\#8}{\lst@lExtend\lst@delim{\csname\@lst @#2DM@\lst@delimtype \expandafter\endcsname\lst@arg}}\@ifundefined{\@lst @#2DM@\lst@delimtype}{{\lst@lExtend\lst@delim{\csname\@lst @#2\@\lst@delimtype \expandafter\endcsname\lst@arg}}}{{\lst@lExtend\lst@delim{\csname\@lst @#2DM@\lst@delimtype \expandafter\endcsname\lst@arg}}}\@ifnextchar[\lst@Delim@delall\lst@Delim@delim}
\else
\if\relax#4\@nil\else\edef\lst@delimstyle{\expandafter\noexpand\csname\@lst @#2DM@\lst@delimtype\endcsname} \fi
\fi
\fi
\expandafter\lst@IfOneOf\lst@delimtype\relax#3\@ifundefined{\@lst @#2DM@\lst@delimtype}{{\lst@lExtend\lst@delim{\csname\@lst @#2\@\lst@delimtype \expandafter\endcsname\lst@arg}}}{{\lst@lExtend\lst@delim{\csname\@lst @#2DM@\lst@delimtype \expandafter\endcsname\lst@arg}}}\@ifnextchar[\lst@Delim@delall\lst@Delim@delim}

\lst@Delim@delim

If the delimiter is not empty, we convert the delimiter and append it to \lst@arg. Ditto \lst@Begin..., \lst@End..., and the style and mode selection.
\expandafter\lst@Begin\lst@Delim@args\expandafter
{\lst@delimtype{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}}\@ifundefined{\@lst @#2DM@\lst@delimtype}{\lst@lExtend\lst@delim{\csname\@lst @#2\@\lst@delimtype \expandafter\endcsname\lst@arg}}{\lst@lExtend\lst@delim{\csname\@lst @#2DM@\lst@delimtype \expandafter\endcsname\lst@arg}}\@ifnextchar[\lst@Delim@delall\lst@Delim@delim}

\lst@Delim@delim

If the type is known, we either choose dynamic or static mode and use the contents of \lst@arg as arguments. All this is put into \lst@delim.
\let\lst@delim\@empty\expandafter\lst@IfOneOf\lst@delimtype\relax\#3\@ifundefined{\@lst @#2DM@\lst@delimtype}{\lst@lExtend\lst@delim{\csname\@lst @#2\@\lst@delimtype \expandafter\endcsname\lst@arg}}{\lst@lExtend\lst@delim{\csname\@lst @#2DM@\lst@delimtype \expandafter\endcsname\lst@arg}}\@ifnextchar[\lst@Delim@delall\lst@Delim@delim}
\expandafter\lst@End\lst@Delim@args\expandafter
{\lst@delimtype{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}}\@ifundefined{\@lst @#2DM@\lst@delimtype}{\lst@lExtend\lst@delim{\csname\@lst @#2\@\lst@delimtype \expandafter\endcsname\lst@arg}}{\lst@lExtend\lst@delim{\csname\@lst @#2DM@\lst@delimtype \expandafter\endcsname\lst@arg}}\@ifnextchar[\lst@Delim@delall\lst@Delim@delim}
Now, depending on the mode \#4 we either remove this particular delimiter or append it to all current ones.

\begingroup
\lst@false \let\lst@next\lst@XConvert
\@ifnextchar #4{\xdef\lst@delimtype{\expandafter\@gobble\lst@delimtype}
\lst@lAddTo\lst@arg{\@empty#5}\
\lst@GobbleNil}
\let\lst@arg\@gtempa
\ifx #7\@nil
\expandafter\lst@Delim@args@\expandafter{\lst@delimstyle}\
\fi}
\endgroup

Recall that the style is ‘selected’ by \def\lst@currstyle#5, and this ‘argument’ \#5 is to be added now. Depending on the settings at the very beginning, we use either {\meta{style}}\lst@modetrue—which selects the style and deactivates keyword detection—, or {\meta{style}}—which defines an empty style macro and executes the style for cumulative styles—, or {\meta{style}}—which just
defines the style macro. Note that we have to use two extra group levels below:
one is discarded directly by \lst@lAddTo and the other by \lst@DelimDM[@(type)].

\lst@Delim@del To delete a particular delimiter, we iterate down the list of delimiters and compare
the current item with the user supplied.

\lst@Delim@delall To delete a whole class of delimiters, we first expand the control sequence name,
init some other data, and call a submacro to do the work.

We first discard a preceding \lst@UseDynamicMode.

Then we can check whether (the following) \lst@(delimiter name)... matches the
delimiter class given by \lst@delim.
Here we put the arguments together to fit \lst@CDef. Note that the very last argument \@empty to \lst@CDef is a brake for \lst@CArgEmpty and \lst@DelimOpen.

After a bug report from Vespe Savikko I added braces around #7.

This service macro will actually define all string delimiters.

\lst@delimtypes is the list of general delimiter types.

\lst@DelimKey We just put together the arguments for \lst@Delim.
1429 \lst@Delim()#2\relax
1430 \{Delim\}\lst@delimtypes #1%
1431 {\lst@BeginDelim\lst@EndDelim}
1432 \i\empty\{\lst@BeginIDelim\lst@EndIDelim\}

\lst@DelimDM@l all use \lst@DelimKey.
moredelim 1433 \lst@Key{delim}\relax{\lst@DelimKey\empty{#1}}
deletedelim 1444 \lst@Key{moredelim}\relax{\lst@DelimKey\relax{#1}}
1445 \lst@Key{deletedelim}\relax{\lst@DelimKey\nil{#1}}

\lst@DelimDM@l Nothing special here.
\lst@DelimDM@s

\lst@DelimDM@l 1436 \gdef\lst@DelimDM@l#1#2\@empty#3#4#5{%
1437 \lst@CArg #2\relax\lst@DefDelimB{}{}{}#3{#1}{#5\lst@Lmodetrue}}
1438 \gdef\lst@DelimDM@s#1#2#3\@empty#4#5#6{%
1439 \lst@CArg #2\relax\lst@DefDelimE{}{}{}#4{#1}{#6}%
1440 \lst@CArg #3\relax\lst@DefDelimE{}{}{}#5{#1}}

\langle /kernel \rangle

15.4.1 Strings
Just starting a new aspect.
\lst@BeginAspect{strings}
\lst@stringtypes
is the list of \ldots\ string types? The several valid string types are defined on page 47.
\lst@AddToHook{EmptyStyle}{\let\lst@stringstyle\@empty}
\lst@stringstyle
You shouldn’t need comments on the following two lines, do you?
\lst@Key{stringstyle}{}{\def\lst@stringstyle{#1}}
\lst@AddToHook{EmptyStyle}{\let\lst@stringstyle\@empty}
\lst@BeginString
\lst@BeginString
Note that the tokens after \lst@DelimOpen are arguments! The only special here is that we switch to ‘keepspaces’ after starting a string, if necessary. A bug reported by Vespe Savikko has gone due to the use of \lst@DelimOpen.
\lst@BeginString
\lst@ifexstrings\else
\lst@ifshowstringspaces
\lst@keepspacestrue
\let\lst@outputspace\lst@visiblespace
\fi}
\lst@AddToHookExe{ExcludeDelims}{\let\lst@ifexstrings\iffalse}
\lst@EndString

Again the two tokens following \lst@DelimClose are arguments.
\gdef\lst@EndString{\lst@DelimClose\lst@ifexstrings\else}

And now all the \lst@StringDM@⟨type⟩ definitions.
\gdef\lst@StringDM@d{\lst@StringDM@d}
\lst@StringDM@d
'd' means no extra work.; the first three arguments after \lst@DefDelimBE are left empty. The others are used to start and end the string.
\gdef\lst@StringDM@d#1#2\@empty#3#4#5{\let\lst@next\@gobblethree
\lst@CArg #2\relax\lst@DefDelimBE{}{}{}#3{#1}{#5}#4}
\lst@StringDM@b
The \lst@ifletter\ldots\fi has been inserted after bug reports by Daniel Gerigk and Peter Bartke. If the last other character is a backslash (4th line), we gobble the 'end string' token sequence.
\gdef\lst@StringDM@b#1#2\@empty#3#4#5{\let\lst@ifbstring\iftrue
\let\lst@ifbstring\iffalse
\lst@CArg #2\relax\lst@DefDelimBE{\lst@ifletter \lst@Output \lst@letterfalse \fi}\
\ifx\lst@lastother\lstum@backslash\expandafter\@gobblethree\fi
{#3}{#1}{#5}{#4}
\global\let\lst@ifbstring\iffalse % init
Heiko Heil reported problems with double backslashes. So:
\gdef\lst@StringDM@bd\lst@StringDM@db\lst@StringDM@db
\global\let\lst@StringDM@bd\lst@StringDM@b
\global\let\lst@StringDM@db\lst@StringDM@bd
\lst@StringDM@bd
is just the same and the same as \lst@StringDM@b.
\lst@StringDM@bd
\global\let\lst@StringDM@bd\lst@StringDM@b
\global\let\lst@StringDM@bd\lst@StringDM@bd
\lst@StringDM@bd
is for Matlab. We enter string mode only if the last character is not in the following list of exceptional characters: letters, digits, period, quote, right parenthesis, right bracket, and right brace. The first list has been extended after bug reports from Christian Kindinger, Benjamin Schubert, and Stefan Stoll.
\gdef\lst@StringDM@m#1#2\@empty#3#4#5{\let\lst@next\@gobblethree
\lst@CArg #2\relax\lst@DefDelimBE{\lst@ProcessOther\lstum@backslash\
\lst@ProcessOther\lstum@backslash
\let\lst@lastother\relax}\
{#3}{#1}{#5}{#4}
\global\let\lst@StringDM@m\lst@StringDM@b
133
\lst@StringDM@ is for string-delimited strings, just as for comments. This is needed for Ruby, and possibly other languages.

\lst@String@mf

It remains to define this macro. In contrast to \lst@PrintDelim, we don’t use \lst@modetrue to allow keyword detection here.
15.4.2 Comments

That’s what we are working on.

\lst@BeginAspect{comments}

\lst@commentmode is a general purpose mode for comments.

\lst@NewMode\lst@commentmode

\lst@commenttypes Via \texttt{comment} available comment types: line, fixed column, single, and nested and all with preceding \texttt{i} for invisible comments.

\gdef\lst@commenttypes{l,f,s,n}

\lst@CommentKey We just put together the arguments for \texttt{\lst@Delim}.

\gdef\lst@CommentKey#1#2{\lst@Delim\texttt{\lst@commentstyle #2}\relax\{Comment}\lst@commenttypes #1\%\lst@BeginComment\lst@EndComment\i\@empty{\lst@BeginInvisible\lst@EndInvisible}}

\texttt{comment} The keys are easy since defined in terms of \texttt{\lst@CommentKey}.

\texttt{morecomment} 

\gdef\lst@CommentKey{\relax{\lst@CommentKey\Empty\{#1\}}}

\texttt{deletecomment} 

\gdef\lst@CommentKey{\relax{\lst@CommentKey\relax\{#1\}}}

\texttt{commentstyle} Any hints necessary?

\gdef\lst@CommentKey{\{#1\}}

\lst@BeginComment \lst@EndComment Once more the three tokens following \texttt{\lst@DelimOpen} are arguments.

\gdef\lst@BeginComment{\%\lst@DelimOpen\lst@ifexcomments\else\lst@BeginComment\lst@EndComment\i\@empty{\lst@BeginInvisible\lst@EndInvisible}}

\texttt{Ditto.}

\gdef\lst@EndComment{\lst@DelimClose\lst@ifexcomments\else}

\lst@AddToHook{AfterBeginComment}{\lst@AddToHook{EmptyStyle}{\lt@style}}

\lst@BeginComment\lst@EndComment
Now we provide all \lst@CommentDM@[type] macros.

\lst@CommentDM@l is easy—thanks to \lst@CArg and \lst@DefDelimB. Note that the ‘end comment’ argument #4 is not used here.

\lst@CommentDM@f is slightly more work. First we provide the number of preceding columns.

\lst@CommentDM@s Nothing special here.

\lst@CommentDM@n We either give an error message or define the nested comment.
15.4.3 PODs

PODs are defined as a separate aspect.

printpod  We begin with the user keys, which I introduced after communication with Michael Piotrowski.

podcomment  \lst@BeginAspect{pod}

\lst@PODmode  is the static mode for PODs.

\lst@NewMode\lst@PODmode

We adjust some characters if the user has selected podcomment=true.

\lst@AddToHook{SelectCharTable}{\lst@ifpodcomment
\lst@CArgX =\relax\lst@DefDelimB{}{}

The following code is executed if we've found an equality sign and haven't entered a mode (in fact if mode changes are allowed): We ‘begin drop output’ and gobble the usual begin of comment sequence (via \@gobblethree) if PODs aren’t be printed. Moreover we gobble it if the current column number is not zero—\@tempcnta is valued below.

\ifnum\@tempcnta=\z@
\lst@ifprintpod\else
\def\lst@beginDropOutput\lst@BeginDropOutput\lst@PODmode\%
\expandafter\expandafter\expandafter\@gobblethree
\fi
\else
\expandafter\@gobblethree
\fi
\lst@BeginComment\lst@PODmode{{\lst@commentstyle}}

If we come to \textasciitilde, we calculate the current column number (zero based).

\lst@CArgX =cut\textasciitilde\relax\lst@DefDelimE
\lst@CalcColumn%

If there is additionally cut+EOL and if we are in \lst@PODmode but not in column one, we must gobble the ‘end comment sequence’.

\ifnum\@tempcnta=\z@
\expandafter\@gobblethree
\fi%
\}
\lst@EndComment\lst@PODmode
\fi

\lst@EndAspect


\lst@EndAspect
15.4.4 Tags

Support for HTML and other 'markup languages'.

\lst@BeginAspect{keywords}{html}

\lst@tagtypes Again we begin with the list of tag types. It's rather short.
\gdef\lst@tagtypes{a}

\lst@TagKey Again we just put together the arguments for \lst@Delim and ...
\gdef\lst@TagKey#1#2{%
\lst@Delim\lst@tagstyle #2\relax
\lst@tagtypes #1%
\lst@BeginTag\lst@EndTag%
@@end\@empty{}}

tag ... we use the definition here.
\lst@Key{tag}\relax{\lst@TagKey\@empty{#1}}
tagstyle You shouldn't need comments on the following two lines, do you?
\lst@Key{tagstyle}{}{\def\lst@tagstyle{#1}}
\lst@AddToHook{EmptyStyle}{\let\lst@tagstyle\@empty}

\lst@BeginTag The special things here are: (1) We activate keyword detection inside tags and (2) we initialize the switch \lst@iffirstintag if necessary.
\gdef\lst@BeginTag{%
\lst@DelimOpen
\lst@ifextags\else
\lst@ifkeywords\iftrue
\lst@ifmarkfirstintag \lst@firstintagtrue \fi}
\lst@AddToHookExe{ExcludeDelims}{\let\lst@ifextags\iffalse}
\lst@EndTag is just like the other \lst@End{whatever} definitions.
\gdef\lst@EndTag{\lst@DelimClose\lst@ifextags\iffalse}

usekeywordsintag The second key has already been 'used'.
markfirstintag \lst@Key{usekeywordsintag}t{t}{\lstKV@SetIf{#1}\lst@ifusekeysintag}
\lst@Key{markfirstintag}f{t}{\lstKV@SetIf{#1}\lst@ifmarkfirstintag}

For this, we install a (global) switch, ...
\gdef\lst@firstintagtrue{\global\let\lst@iffirstintag\iftrue}
\gdef\lst@iffirstintag\iffalse
\global\let\lst@iffirstintag\iffalse

... which is reset by the output of an identifier but not by other output.
\lst@AddToHook{PostOutput}{\lst@tagresetfirst}
\lst@AddToHook{Output}{
\lst@tagresetfirst\iffalse}
\lst@AddToHook{OutputOther}{\lst@tagresetfirst{}}

Now we only need to test against this switch in the Output hook.
\lst@AddToHook{Output}{
\lst@iffirstintag \let\lst@thestyle\lst@gkeywords@sty \fi}
Moreover we check here, whether the keyword style is always to be used.

\lst@tagmode We allocate the mode and ... 

deactivate keyword detection if any tag delimiter is defined (see below).

\lst@Tag@s The definition of the one and only delimiter type is not that interesting. Compared with the others we set \lst@ifnotag and enter tag mode only if we aren’t in tag mode.

\lst@BeginCDATA This macro is used by the XML language definition.

15.5 Replacing input

\lst@ReplaceInput is defined in terms of \lst@CArgX and \lst@CDefX.

literate Jason Alexander asked for something like that. The key looks for a star and saves the argument.

Internally we don’t make use of the ‘replace input’ feature any more.
Note that we check `\lst@OutputBox` for being `\@gobble`. This is due to a bug report by Jared Warren.

\lst@BeginDropInput  We deactivate all ‘process’ macros. `\lst@modetrue` does this for all up-coming string delimiters, comments, and so on.

15.6 Escaping to \LaTeX{}x

We now define the ... damned ... the aspect has escaped!

texcl Communication with Jörn Wilms is responsible for this key. The definition and the first hooks are easy.
If the user wants \TeX{} comment lines, we print the comment separator and interrupt the normal processing.

```latex
\lst@AddToHook{AfterBeginComment}
\{\lst@ifextra \lst@ifmode \lst@ifdropinput\else
  \lst@PrintToken \\
  \lst@LeaveMode \lst@InterruptModes \\
  \lst@EnterMode{\lst@TeXmode}{\lst@modetrue\lst@commentstyle}\\
  \expandafter\expandafter\expandafter\lst@escapebegin \\
  \fi \fi \fi}
\lst@NewMode\lst@TeXXmode

\lst@ActiveCDefX Same as \lst@ActiveCDefX but we both make \#1 active and assign a new catcode.
\lst@ActiveCDefX@#1#2#3{
  \catcode'#1\active\lccode'\~=#1%%
  \lowercase{\lst@CDefIt~}{#2}{#3}{}%}

\lst@Escape gets four arguments all in all. The first and second are the ‘begin’ and ‘end’ escape sequences, the third is executed when the escape starts, and the fourth right before ending it. We use the same mechanism as for \TeX{} comment lines. The \lst@ifdropinput test has been added after a bug report by Michael Weber. The \lst@newlines\z@ was added due to a bug report by Frank Atanassow.

\lst@Escape#1#2#3#4{%
  \lst@CArgX #1\relax\lst@CDefX
  {}%
  {\lst@ifdropinput\else
    \lst@TrackNewLines\lst@OutputLostSpace \lst@XPrintToken \\
    \lst@InterruptModes \\
    \lst@EnterMode{\lst@TeXmode}{\lst@modetrue}%
  \fi \fi \fi}%

Now we must define the character sequence to end the escape.

```
\lst@CArg #2\relax\lst@ActiveCDefX
{}%
\{\lst@escapeend #4\lst@LeaveAllModes\lst@ReenterModes\%
\{\lst@MProcessListing\%
\else
\lst@CArg #2\relax\lst@ActiveCDefX
{}%
\{\lst@escapeend #4\lst@LeaveAllModes\lst@ReenterModes
\{\lst@newlines\z@ \lst@whitespacefalse}\%
\}%
\fi
\#3\lst@escapebegin
\fi%
{}}%
```

The \lst@whitespacefalse above was added after a bug report from Martin Steffen.

```
\lst@NewMode\lst@TeXXmode
```

\#3\lst@escapebegin
\fi%
{}}%
```

The \lst@whitespacefalse above was added after a bug report from Martin Steffen.

```
\lst@NewMode\lst@TeXXmode
```

\lst@Escape#1\relax\lst@ActiveCDefX
{}%
\{\lst@escapebegin #1\}
\lst@Key{escapebegin}{\def\lst@escapebegin{#1}}
\lst@Key{escapeend}{\def\lst@escapeend{#1}}
escapechar The introduction of this key is due to a communication with Rui Oliveira. We define \lst@DefEsc and execute it after selecting the standard character table.

\begin{verbatim}
\lst@Key{escapechar}{\{}
  \{\ifx\@empty#1\@empty
    \let\lst@DefEsc\relax
  \else
    \def\lst@DefEsc{\lst@Escape{#1}{#1}{}{}}%
  \fi}
\lst@AddToHook{TextStyle}{\let\lst@DefEsc\@empty}
\lst@AddToHook{SelectCharTable}{\lst@DefEsc}
\end{verbatim}

escapeinside Nearly the same.

\begin{verbatim}
\lst@Key{escapeinside}{}{\lstKV@TwoArg{#1}%
  \{\let\lst@DefEsc\@empty
    \ifx\@empty##1@empty\else \ifx\@empty##2\@empty\else
      \def\lst@DefEsc{\lst@Escape{##1}{##2}{}{}}%
    \fi\fi
  \fi{fi}}
\end{verbatim}

mathescape This is a switch and checked after character table selection. We use \lst@Escape with math shifts as arguments, but all inside \hbox to determine the correct width.

\begin{verbatim}
\lst@Key{mathescape}{false}[t]{\lstKV@SetIf{#1}\lst@ifmathescape
\lst@AddToHook{SelectCharTable}{\lst@ifmathescape \lst@Escape{$}{\$}%
  \setbox\@tempboxa=\hbox\bgroup\$}
  \egroup \lst@CalcLostSpaceAndOutput}\fi}
\end{verbatim}

\lst@EndAspect

\begin{verbatim}
\langle/
\end{verbatim}

16 Keywords

16.1 Making tests

We begin a new and very important aspect. First of all we need to initialize some variables in order to work around a bug reported by Beat Birkhofer.

\begin{verbatim}
\lst@BeginAspect{keywords}
\global\let\lst@ifsensitive\iftrue % init
\global\let\lst@ifsensitivedefed\iffalse % \global
\end{verbatim}

All keyword tests take the following three arguments.

- \#1 = \langle prefix\rangle
- \#2 = \lst@⟨name⟩@list (a list of macros which contain the keywords)
- \#3 = \lst@⟨name⟩@sty (global style macro)

We begin with non memory-saving tests.

\begin{verbatim}
\lst@KeywordTest
\end{verbatim}

Fast keyword tests take advance of the \lst@UM construction in section 15.3. If \lst@UM is empty, all 'use macro' characters expand to their original characters. Since \lst⟨prefix⟩⟨keyword⟩ will be equivalent to the appropriate style, we only
need to build the control sequence \lst\{prefix\}@\{current token\} and assign it to \lst@thestyle.

\lst@KEYWORDTEST Case insensitive tests make the current character string upper case and give it to a submacro similar to \lst@KeywordTest.

\lst@WORKINGTEST The same except that \lst\{prefix\}@\{current token\} might be a working procedure; it is executed.

\lst@DefineKeywords Eventually we need macros which define and undefine \lst\{prefix\}@\{keyword\}.

Here the arguments are

#1 = \{prefix\}
#2 = \lst\{name\} \{a keyword list\}
#3 = \lst@g\{name\}@sty

We make the keywords upper case if necessary, ...
\lst@ifsensitive
\def\lst@next{\lstfor#2}\else
\def\lst@next{\uppercase\expandafter{\expandafter\lstfor#2}}\fi
\lst@next\do
... iterate through the list, and make \lst{(prefix)@⟨keyword⟩} (if undefined) equivalent to \lst@g{⟨name⟩}@sty which is possibly a working macro.
{\expandafter\ifx\csname\@lst#1@##1\endcsname\relax
\global\expandafter\let\csname\@lst#1@##1\endcsname#3\fi}}
\lst@UndefineKeywords
We make the keywords upper case if necessary, ...
\gdef\lst@UndefineKeywords#1#2#3{\lst@ifsensitive\def\lst@next{\lstfor#2}\else\def\lst@next{\uppercase\expandafter{\expandafter\lstfor#2}}\fi
\lst@next\do
... iterate through the list, and ‘undefine’ \lst{(prefix)@⟨keyword⟩} if it’s equivalent to \lst@g{⟨name⟩}@sty.
{\expandafter\ifx\csname\@lst#1@##1\endcsname#3\global\expandafter\let\csname\@lst#1@##1\endcsname\relax
\fi}}

Thanks to Magnus Lewis-Smith a wrong #2 in the replacement text could be changed to #3.

And now memory-saving tests.
\fi
\lst@if savemem
\lst@IfOneOutOf
The definition here is similar to \lst@IfOneOf, but its second argument is a \lst@⟨name⟩@list. Therefore we test a list of macros here.
\gdef\lst@IfOneOutOf#1\relax#2{\def\lst@temp##1,#1,##2##3\relax{\ifx\@empty##2\else \expandafter\lst@IOOOfirst \fi}\def\lst@next{\lst@IfOneOutOf@#1\relax}\expandafter\lst@next#2\relax\relax}

We either execute the ⟨else⟩ part or make the next test.
\gdef\lst@IfOneOutOf#1\relax#2{\ifx#2\relax\expandafter\lst@IfOneOutOf@#1\relax#2\relax\else \expandafter\lst@temp\expandafter,#2,#1,\@empty\relax\expandafter\lst@next\fi}\gdef\lst@IfOneOutOf#1\relax#2\relax\fi}
\ifx\iffalse\else\fi
\gdef\lst@IfOneOutOf#1\relax#2\relax#3{\fi#2}

The line \ifx\iffalse\else\fi balances the \fi inside \lst@IOOOfirst.
As in \lst@IFONEOF we need two \uppercase{}s here.
\lst@KWTest{} is a helper for the keyword and working identifier tests. We expand the token and call \lst@IFONEOF. The tests below will append appropriate ⟨then⟩ and ⟨else⟩ arguments.

\lst@MakeMacroUppercase{} makes the contents of #1 (if defined) upper case.
16.2 Installing tests

\lst@InstallTest The arguments are

#1 = ⟨prefix⟩
#2 = \lst@(name)@list
#3 = \lst@(name)
#4 = \lst@g(name)@list
#5 = \lst@g(name)
#6 = \lst@g(name)@sty
#7 = w|s (working procedure or style)
#8 = d|o (DetectKeywords or Output hook)

We just insert hook material. The tests will be inserted on demand.

\gdef\lst@InstallTest#1#2#3#4#5#6#7#8{%
  \lst@AddToHook{TrackKeywords}{\lst@TrackKeywords{#1}#2#4#6#7#8}%
  \lst@AddToHook{PostTrackKeywords}{\lst@PostTrackKeywords#2#3#4#5}%
}\lst@AddToHook{Init}{\lsthk@TrackKeywords\lsthk@PostTrackKeywords}
\lst@AddToHook{TrackKeywords}
\global\let\lst@DoDefineKeywords\@empty
\lst@AddToHook{PostTrackKeywords}
\global\let\lst@DoDefineKeywords\@empty
\global\let\lst@ifkeywords\iffalse
\lst@AddToHook{Output}{\lst@ifkeywords \lsthk@DetectKeywords \fi}
\lst@AddToHook{DetectKeywords}{}\global\let\lst@ifkeywords\iftrue
\lst@AddToHookExe{Init}{\let\lst@ifkeywords\iffalse}
\lst@InstallTestNow actually inserts a test.

#1 = ⟨prefix⟩
#2 = \lst@(name)@list
#3 = \lst@g(name)@sty
#4 = w|s (working procedure or style)
#5 = d|o (DetectKeywords or Output hook)

For example, #4#5=sd will add \lst@KeywordTest{⟨prefix⟩} \lst@g(name)@sty to the DetectKeywords hook.

\gdef\lst@InstallTestNow#1#2#3#4#5{%
  \@ifundefined{\string#2#1}{\global\@namedef{\string#2#1}{}%
    \edef\@tempa{\noexpand\lst@AddToHook{\ifx#5dDetectKeywords\else Output\fi}{\ifx #4w\noexpand\lst@WorkingTest \else \noexpand\lst@KeywordTest \fi{#1}\noexpand#2\noexpand#3}}%
}\lst@ifsavemem\@tempa%
\@ifundefined{\@lst#1@if@ins}{}
\lst@ifsavemem
\@temps\@temps
\else
  \@ifundefined{\lst#1@ifsavemem}{}
\endinput

If we are advised to save memory, we insert a test for each ⟨name⟩. Otherwise we install the tests according to ⟨prefix⟩.

\lst@ifsavemem
\@temps
\else
  \@ifundefined{\@lst#1@ifsavemem}{}
Now it gets a bit tricky. We expand the class list \lst@TK{(prefix)}@list behind \lst@TK{(prefix)}@list and use two relaxes as terminators. This will define the keywords of all the classes as keywords of type \langle prefix \rangle. More details come soon.

And nearly the same to undefine all out-dated keywords, which is necessary only if we don't save memory.

Finally we install the keyword test if keywords changed, in particular if they are defined the first time. Note that \lst@InstallTestNow inserts a test only once.

Back to the current keywords. Global macros \lst@g{id} contain globally defined keywords, whereas \lst@{id} contain the true keywords. This way we can keep track of the keywords: If keywords or sensitive changed, we undefine the old (= globally defined) keywords and define the true ones. The arguments of \lst@TK{prefix} are

| #1 = \langle prefix \rangle |
| #2 = \lst@g{name}@sty |
| #3 = \lst@{id} |
| #4 = \lst@g{id} |

Thanks to Holger Arndt the definition of keywords is now delayed via \lst@DoDefineKeywords.
We don’t define and undefine keywords if we try to save memory. But we possibly need to make them upper case, which again wastes some memory.

Reaching the end of the class list, we end the loop.

Here now we undefine the out-dated keywords. While not reaching the end of the global list, we look whether the keyword class #4#5 is still in use or needs to be undefined. Our arguments are

1. \lst@classoffset{#1} = ⟨prefix⟩
2. \lst@name{#2} = ⟨name⟩
3. \lst@style{#3} = ⟨style name⟩
4. \lst@id{#4} = ⟨id⟩
5. \lst@id{#5} = ⟨id⟩

Keywords are up-to-date after InitVars.

\lst@PostTrackKeywords After updating all the keywords, the global keywords and the global list become equivalent to the local ones.

16.3 Classes and families

classoffset just stores the argument in a macro.

Recall the parameters

1. \lst@classoffset{#1} = ⟨prefix⟩
2. \lst@name{#2} = ⟨name⟩
3. \lst@style{#3} = ⟨style name⟩
4. \lst@style{#4} = ⟨style init⟩
First we define the keys and the style key \langle style name \rangle if and only if the name is not empty.

Now we check whether \langle working procedure \rangle is empty. Accordingly we use working procedure or style in the \langle data \rangle definition. The working procedure is defined right here if necessary.

Nothing else is defined here, all the rest is done on demand.
\lst@ProvideFamily provides the member \texttt{\the\@tempcnta} of the family \texttt{#1}. We do nothing if the member already exists. Otherwise we expand the data macro defined above. Note that we don’t use the counter if it equals one. Since a bug report by Kris Luyten keyword families use the prefix \texttt{lstfam} instead of \texttt{lst}. The marker \lstfam@#1\texttt{[\textit{number}]} is defined globally since a bug report by Edsko de Vries.

Now we have the following arguments

\begin{itemize}
\item \texttt{#1} = (prefix)
\item \texttt{#2} = (name)
\item \texttt{#3} = (style name)
\item \texttt{#4} = (default style name)
\item \texttt{#5} = l|o (language or other key)
\item \texttt{#6} = w|s (working procedure or style)
\item \texttt{#7} = d|o (DetectKeywords or Output hook)
\item \texttt{#8} = \texttt{\ifnum\@tempcnta=\@ne\else \the\@tempcnta \fi}
\end{itemize}

We define \texttt{\lst@g\textit{name}\texttt{[number]}@sty} to call either \texttt{\lst@g\textit{name}\texttt{[number]}@wp} with the number as argument or \texttt{\lst@\textit{style name}\texttt{[number]} where the number belongs to the control sequence. We ensure the existence of the style macro. This is done in the \texttt{Init} hook by assigning the default style if necessary.

We call a submacro to do the rest. It requires some control sequences.
Now we have (except that \texttt{number} is possibly always missing)
\begin{align*}
#1 &= \texttt{\lst@⟨name⟩⟨number⟩@list} \\
#2 &= \texttt{\lst@⟨name⟩⟨number⟩} \\
#3 &= \texttt{\lst@⟨name⟩⟨number⟩@also} \\
#4 &= \texttt{\lst@g⟨name⟩⟨number⟩@list} \\
#5 &= \texttt{\lst@g⟨name⟩⟨number⟩} \\
#6 &= \texttt{\lst@g⟨name⟩⟨number⟩@sty} \\
#7 &= \langle \texttt{prefix} \rangle \\
#8 &= \texttt{l|o (language or other key)} \\
#9 &= \texttt{w|s (working procedure or style)} \\
#10 &= \texttt{d|o (DetectKeywords or Output hook)}
\end{align*}

Note that \#9 and \#10 are read by \texttt{\lst@InstallTest}. We initialize all required ‘variables’ (at \texttt{SetLanguage}) and install the test (which definition is in fact also delayed).
\begin{verbatim}
\gdef\lst@ProvideFamily@@#1#2#3#4#5#6#7#8{%
\gdef#1{#2#5}\global\let#2\@empty \global\let#3\@empty % init \gdef#4{#2#5}\global\let#5\@empty % init \if #8l\relax \lst@AddToHook{SetLanguage}{\def#1{#2#5}\let#2\@empty}\fi \lst@InstallTest{#7}#1#2#4#5#6}
\end{verbatim}

\lst@InstallKeywords Now we take advantage of the optional argument construction above. Thus, we just insert \texttt{[⟨one⟩]} as \texttt{⟨number⟩} in the definitions of the keys.
\begin{verbatim}
\gdef\lst@InstallKeywords#1#2#3#4#5{% \lst@Key{#2}\relax {\lst@UseFamily{#2}[⟨one⟩]##1\relax\lst@MakeKeywords}\lst@Key{more#2}\relax {\lst@UseFamily{#2}[⟨one⟩]##1\relax\lst@MakeMoreKeywords}\lst@Key{delete#2}\relax {\lst@UseFamily{#2}[⟨one⟩]##1\relax\lst@DeleteKeywords}\ifx\@empty#3\@empty\else \lst@Key{#3}{#4}{\@namedef{lst@#3}{##1}}\fi \expandafter\lst@InstallFamily\csname\@lst@#2@data\endcsname \csname\@lst@#5\endcsname {#1}{#2}{#3}}
\end{verbatim}

\lst@ProvideStyle If the style macro \#1 is not defined, it becomes equivalent to \#2.
\begin{verbatim}
\gdef\lst@ProvideStyle#1#2{% \ifx#1\@undefined \let#1#2\% \else\ifx#1\relax \let#1#2\fi\fi \expandafter\lst@ProvideFamily\csname\@lst@#1@data\endcsname {#1}{#2}{#3}}
\end{verbatim}

Finally we define \texttt{\lst@MakeKeywords}, \ldots, \texttt{\lst@DeleteKeywords}. We begin with two helper.

\lst@BuildClassList After \#1 follows a comma separated list of keyword classes terminated by \texttt{,\relax}, \eg \texttt{keywords2, emph1, \relax,}. For each \langle item \rangle in this list we append the two macros \texttt{\lst@⟨item⟩\lst@g⟨item⟩} to \#1.
\lst@DeleteClassesIn \ deletes pairs of tokens, namely the arguments #2#3 to the submacro.
\begin{verbatim}
\def\lst@temp##1#2#3##2{\lst@lAddTo#1{##1}##2}##2\let\@tempa#1\let#1\@empty\expandafter\lst@temp\@tempa#2#3\relax\fi
\end{verbatim}
\lst@DeleteKeywords \ We convert the keyword arguments via \lst@MakeKeywords and remove the classes and keywords.
\begin{verbatim}
\def\lst@MakeKeywords[#1]#2#3#4#5#6{\def#3{#4#6}\let#4\@empty\let#5\@empty\lst@MakeMoreKeywords[#1]{#2}#3#4#5#6}
\end{verbatim}
\lst@MakeMoreKeywords \ We append classes and keywords.
\begin{verbatim}
\gdef\lst@MakeMoreKeywords[#1]#2#3#4#5#6{\lst@BuildClassList#3#1,\relax,\lst@DefOther\lst@temp{,#2}\lst@lExtend#4\lst@temp}
\end{verbatim}
\lst@DeleteKeywords \ We convert the keyword arguments via \lst@MakeKeywords and remove the classes and keywords.
\begin{verbatim}
\def\lst@DeleteKeywords[#1]#2#3#4#5#6{\lst@MakeKeywords[#1]#2#3#4#5#6\lst@DeleteClassesIn\lst@tempa\lst@tempb#5#6\lst@DeleteKeysIn#4\lst@temp#2\relax\lst@tempa\lst@tempb#2\relax
\lst@DeleteOther\lst@temp{,#2}\lst@lExtend#4\lst@temp}
\end{verbatim}
\lst@DeleteOther \ We remove empty classes.
\begin{verbatim}
\def\lst@DeleteOther\lst@temp{\ifx\relax#2\@empty\else\expandafter\lst@lExtend\expandafter#1\expandafter{\csname\@lst@g#2\endcsname}{\csname\@lst@#2\endcsname}#2\fi\expandafter\lst@DeleteClassesIn\expandafter#1\fi}
\end{verbatim}

\lst@DeleteClassesIn deletes pairs of tokens, namely the arguments #2#3 to the submacro.
16.4 Main families and classes

Keywords

**keywords** Defining the keyword family gets very, very easy.

The following macro sets a keywordstyle, which ...

... is put together here. If we detect a star after the class number, we insert code to make the keyword uppercase.

**ndkeywords** Second order keywords use the same trick as `\lst@InstallKeywords`.

Dr. Peter Leibner reported two bugs: `\lst@UseKeywords` and `##1` became `\lst@UseFamily` and `#1`.

**keywordsprefix** is implemented experimentally. The one and only prefix indicates its presence by making `\lst@prefixkeyword` empty. We can catch this information in the **Output** hook.

153
Thanks to Bradford Chamberlain we now iterate down the list of ‘other keywords’ and make each active—instead of making the whole argument active. We append the active token sequence to \lst@otherkeywords to define each ‘other’ keyword.

\lst@Key{otherkeywords}{}{%
\let\lst@otherkeywords\@empty
\lst@for{#1}\do{% 
\lst@MakeActive{##1}\%
\lst@lExtend\lst@otherkeywords{%
\expandafter\lst@CArg\lst@temp\relax\lst@CDef
{}\lst@PrintOtherKeyword\@empty}}}
\lst@AddToHook{SelectCharTable}{\lst@otherkeywords}

\lst@PrintOtherkeyword has been changed to \lst@PrintOtherKeyword after a bug report by Peter Bartke.

\lst@PrintOtherKeyword print preceding characters, prepare the output and typeset the argument in keyword style. James Willans reported problems when the output routine is invoked within \begingroup and \endgroup. Now the definition is restructured.

\gdef\lst@PrintOtherKeyword#1\@empty{%
\lst@XPrintToken
\begingroup 
\lst@modetrue \lsthk@TextStyle 
\let\lst@ProcessDigit\lst@ProcessLetter 
\let\lst@ProcessOther\lst@ProcessLetter 
\lst@lettertrue 
#1% 
\lst@SaveToken 
\endgroup 
\lst@RestoreToken 
\global\let\lst@savedcurrstyle\lst@currstyle 
\let\lst@currstyle\lst@gkeywords@sty 
\lst@Output 
\let\lst@currstyle\lst@savedcurrstyle}
\lst@EndAspect
\lst@BeginAspect[keywords]{emph}
\lst@InstallFamily e{emph}{emphstyle}{}{emphstyle}{}od
\lst@EndAspect
\lst@BeginAspect[keywords]{tex}

To do: Which part of TextStyle hook is required? Is it required anymore, i.e. after the restruction? Need to move it elsewhere?

The emphasize family

is just one macro call here.

\lst@BeginAspect[keywords]{emph}
\lst@InstallFamily e{emph}{emphstyle}{}{emphstyle}{}od
\lst@EndAspect
\lst@BeginAspect[keywords]{tex}
The style-key checks for the optional star (which must be in front of the optional class argument).

To make the backslash belong to the control sequence, it is merged with the following token. This option was suggested by Morten Høgholm. Christian Schneider pointed out that the original implementation was broken when the identifier was preceded by an “other” character. To fix this (and other bugs), we first output whatever is in the current token before merging.

Compiler directives

First some usual stuff.

directives

The initialization of \lst@directives has been added after a bug report from Kris Luyten.
Now we define a new delimiter for directives: We enter ‘directive mode’ only in the first column.

We introduce a new string type (thanks to R. Isernhagen), which ...

16.5 Keyword comments

includes both comment types and is possibly split into this and dkcs.
2236 \gdef\lst@BeginKC{\%\lst@ResetToken \lst@BeginComment\lst@KCSmode{{\lst@commentstyle}\lst@modetrue}\% \@empty}\%}

(2) we must ensure that the comment starts after printing the comment delimiter since it could be a keyword. We assign \lst@BeginKC[S] to \lst@KCpost, which is executed and reset in PostOutput.

\lst@AddToHook{PostOutput}{\lst@KCpost \global\let\lst@KCpost\@empty}
\global\let\lst@KCpost\@empty

leaves the comment mode before the (temporarily saved) comment delimiter is printed.

\lst@EndKC leaves the comment mode before the (temporarily saved) comment delimiter is printed.

\gdef\lst@EndKC{\lst@SaveToken \lst@LeaveMode \lst@RestoreToken \let\lst@thestyle\lst@identifierstyle \lsthk@Output}

\lst@InstallKeywords{kc}{keywordcomment}{}
\lst@Key{keywordcommentsemicolon}{\lstKV@ThreeArg{#1}%
\def\lst@KCAkeywordsB{##1}%
\def\lst@KCAkeywordsE{##2}%
\def\lst@KCBkeywordsB{##3}%
\def\lst@KCkeywords{##1##2##3}}
\lst@AddToHook{SetLanguage}{\let\lst@KCAkeywordsB\@empty \let\lst@KCAkeywordsE\@empty \let\lst@KCBkeywordsB\@empty \let\lst@KCkeywords\@empty}

We define an appropriate semicolon if this keyword comment type is defined. Appropriate means that we leave any keyword comment mode if active. Oldrich Jedlicka reported a bug and provided the fix, the two \@empty s.

\lst@AddToHook{SelectCharTable}{\ifx\lst@KCkeywords\@empty\else
\lst@DefSaveDef{'\;}'\lsts@EKC
\ifnum\lst@mode=\lst@KCmode \lst@EndComment\@empty \else
\ifnum\lst@mode=\lst@KCSmode \lst@EndComment\@empty
\fi
\fi
\fi}

\keywords{keywordcommentsemicolon} The key simply stores the keywords. After a bug report by Norbert Eisinger the initialization in SetLanguage has been added.
The ‘working identifier’ macros enter respectively leave comment mode.

```latex
\gdef\lst@KCAWorkB{% 
  \lst@ifmode\else \global\let\lst@KCpost\lst@BeginKC \fi}
\gdef\lst@KCBWorkB{% 
  \lst@ifmode\else \global\let\lst@KCpost\lst@BeginKCS \fi}
\gdef\lst@KCAWorkE{\ifnum\lst@mode=\lst@KCmode \lst@EndKC \fi}
```

Now we install the tests and initialize the given macros.

```latex
\lst@ProvideFamily@@
\lst@KCAkeywordsB@list\lst@KCAkeywordsB \lst@KC@also
\lst@gKCAkeywordsB@list\lst@gKCAkeywordsB \lst@KCAWorkB
{kcb}owo % prefix, other key, working procedure, Output hook
\lst@ProvideFamily@@
\lst@KCAkeywordsE@list\lst@KCAkeywordsE \lst@KC@also
\lst@gKCAkeywordsE@list\lst@gKCAkeywordsE \lst@KCAWorkE
{kce}owo
\lst@EndAspect
```

16.6 Export of identifiers

One more ‘keyword’ class.

```latex
\lstindexmacro\lst@BeginAspect[keywords]{index}
\lst@InstallFamily w{index}{indexstyle}\lstindexmacro{indexstyle}
{\csname\@lst@indexstyle#1\expandafter\endcsname \expandafter{\the\lst@token}} od
\lst@UserCommand\lstindexmacro#1{\index{\texttt{#1}}}
\lst@EndAspect
```

The ‘idea’ here is the usage of a global \lst@ifprocname, indicating a preceding ‘procedure keyword’. All the other is known stuff.

```latex
\procesnamestyle \procesnamekeys\lst@BeginAspect[keywords]{pronames}
\gdef\lst@procmnametrue{\global\let\lst@ifprocname\iftrue}
\gdef\lst@procmnamefalse{\global\let\lst@ifprocname\iffalse}
\lst@AddToHook{Init}{\lst@procmnamefalse}
\lst@AddToHook{DetectKeywords}{\lst@ifprocname
{\csname\@lst@gindex@sty\endcsname \lst@thestyle}}
```

The idea here is the usage of a global \lst@ifprocname, indicating a preceding ‘procedure keyword’. All the other is known stuff.
The default definition of `lst@indexstyle` above has been moved outside the hook after a bug report from Ulrich G. Wortmann.

17 More aspects and keys

**basicstyle** There is no better place to define these keys, I think.

**inputencoding**

Michael Niedermair asked for a key like `inputencoding`.

17.1 Styles and languages

We begin with style definition and selection.

**lststylefiles** This macro is defined if and only if it’s undefined yet.

**lstdefinestyle** are defined in terms of `lst@DefStyle`, which is defined via `lst@DefDriver`. 

159
The 'empty' style calls the initial empty hook \texttt{EmptyStyle}.
\begin{verbatim}
global\namedef{lststy@$}{\lsthk@EmptyStyle} \lst@AddToHook{EmptyStyle}{\% init} \end{verbatim}

\texttt{style} is an application of \texttt{\lst@LAS}. We just specify the hook and an empty argument as 'pre' and 'post' code.
\begin{verbatim}
\lst@Key{style}{\% \relax\lst@LAS{style}{sty}{[\{\}]{#1}}\lst@NoAlias\lststylefiles \lsthk@SetStyle \} \} \lst@AddToHook{SetStyle}{\% init} \lst@EndAspect \langle/\misc\rangle

Now we deal with commands used in defining and selecting programming languages, in particular with aliases.
\begin{verbatim}
\lst@BeginAspect{language} \lstlanguagefiles \end{verbatim}

\texttt{language} is mainly an application of \texttt{\lst@LAS}. \texttt{alsolanguage} is also language is mainly an application of \texttt{\lst@LAS}.
\begin{verbatim}
\lst@Key{language}{\% \relax\lst@LAS{language}{lang}{[\{\}]{##1}{##2}}\lst@FindAlias\lstlanguagefiles \lsthk@SetLanguage \} \) \let\lst@language\lst@malias \let\lst@dialect\lst@oalias}

Ditto, we simply don’t execute \lsthk@SetLanguage.
\begin{verbatim}
\lst@Key{alsolanguage}{\% \relax\lst@LAS{alsolanguage}{lang}{[\{\}]{##1}{##2}}\lst@FindAlias\lstlanguagefiles \lsthk@SetLanguage \} \) \let\lst@language\lst@malias \let\lst@dialect\lst@oalias}}

\end{verbatim}

\texttt{\lstalias} Now we concentrate on aliases and default dialects. \texttt{\lsta@{language}\$(\texttt{dialect})} and \texttt{\lsta@{language}} contain the aliases of a particular dialect respectively a complete language. We’ll use a $-character to separate a language name from its
dialect. Thanks to Walter E. Brown for reporting a problem with the argument
delimiter ‘|’ in a previous definition of \lstalias@.

\lst@Key{defaultdialect}
{\lstKV@OptArg{#1}{\lst@NormedNameDef{lstdd@##2}{##1}}}\lst@FindAlias

Now we have to find a language. First we test for a complete language alias, then
we set the default dialect if necessary.

\lst@RequireLanguages This definition will be equivalent to \lstloadlanguages. We requested the given
list of languages and load additionally required aspects.

\lstloadlanguages is the same as \lst@RequireLanguages.
\lst@EndAspect

17.2 Format definitions*

\lst@BeginAspect{formats}
This macro is defined if and only if it’s undefined yet.
\lst@defineformat are defined in terms of \lst@DefFormat, which is defined via \lst@DefDriver. We provide the ‘empty’ format.

format is an application of \lst@LAS. We just specify the hook as ‘pre’ and an empty argument as ‘post’ code.

Helpers Our goal is to define the yet unknown \lst@UseFormat. This definition will parse the user supplied format. We start with some general macros.

\lst@fmtSplit splits the content of the macro \#1 at \#2 in the preceding characters \lst@fmta and the following ones \lst@fmtb. \lst@if is false if and only if \#1 doesn’t contain \#2.

\lst@IfNextCharWhitespace is defined in terms of \lst@IfSubstring.

And here come all white space characters.
\lst@fmtIfIdentifier  tests the first character of #1
\begin{verbatim}
def\lst@fmtIfIdentifier#1{% 
  \ifx\relax#1\@empty 
    \expandafter\@secondoftwo 
  \else 
    \expandafter\lst@fmtIfIdentifier\expandafter@\expandafter#1\fi}
\end{verbatim}

against the ‘letters’ _, @, A, ..., Z and a, ..., z.
\begin{verbatim}
def\lst@fmtIfIdentifier@#1#2\relax{% 
  \let\lst@next\@secondoftwo 
  \ifnum'#1='_\else 
    \ifnum'#1<64\else 
      \ifnum'#1<91\let\lst@next\@firstoftwo\else 
        \ifnum'#1<97\else 
          \fi \fi \fi \fi \fi 
  \lst@next}
\end{verbatim}

\lst@fmtIfNextCharIn  is required for the optional ⟨exceptional characters⟩. The implementation is easy—refer section 13.1.
\begin{verbatim}
def\lst@fmtIfNextCharIn#1{% 
  \ifx\@empty#1\@empty \expandafter\@secondoftwo \else 
    \def\lst@next{\lst@fmtIfNextCharIn@{#1}}% 
  \expandafter\lst@next\fi}
\end{verbatim}

\lst@fmtIfNextCharIn@#1#2#3#4{% 
  \def\lst@temp##1#4##2##3\relax{\ifx@empty##2\expandafter\@secondoftwo 
  \else \expandafter\@firstoftwo \fi} 
  \lst@temp#1#4@empty@\relax{#2}{#3}#4}

\lst@fmtCDef  We need derivations of \lst@CDef  and \lst@CDefX  we have to test the next character against the sequence #5 of exceptional characters. These tests are inserted here.
\begin{verbatim}
def\lst@fmtCDef#1{\lst@fmtCDef@#1}
\end{verbatim}

\lst@fmtCDefX  The same but ‘drop input’.
\begin{verbatim}
def\lst@fmtCDefX#1{\lst@fmtCDefX@#1}
\end{verbatim}
The parser applies \lst@fmtSplit to cut a format definition into items, items into ‘input’ and ‘output’, and ‘output’ into ‘pre’ and ‘post’. This should be clear if you are in touch with format definitions.

\lst@UseFormat Now we can start with the parser.

We assign the rest of the format definition, . . .

To do: Insert \let\lst@arg\@empty \expandafter\lst@XConvert\lst@fmtb\@nil \let\lst@fmtb\lst@arg.

Finally we process the next item if the rest is not empty.

We make \lst@fmtc contain the preceding characters as a braced argument. To add more arguments, we first split the replacement tokens at the control sequence \string.

We append an empty argument or \lst@fmtPre with ‘\string-preceding’ tokens as argument. We do the same for the tokens after \string.
Eventually we extend \lst@fmtformat appropriately. Note that \lst@if still indicates whether the replacement tokens contain \string.

The formatting

\lst@fmtPre
\gdef\lst@fmtPre#1{%\begin{group}\lst@PrintToken\let\newline\lst@fmtEnsureNewLine\let\space\lst@fmtEnsureSpace\let\indent\lst@fmtIndent\let
\noindent\lst@fmtNoindent\end{group}{#1}\end{group}

\lst@fmtPost
\gdef\lst@fmtPost#1{%\global\let\lst@fmtPostOutput\@empty\begingroup\def\newline{\lst@AddTo\lst@fmtPostOutput\lst@fmtEnsureNewLine}\def\space{\aftergroup\lst@fmtEnsurePostSpace}\def\indent{\lst@AddTo\lst@fmtPostOutput\lst@fmtIndent}\def
\noindent{\lst@AddTo\lst@fmtPostOutput\lst@fmtNoindent}\aftergroup\lst@PrintToken\endgroup{#1}\end{group}
To do: This ‘lost space’ doesn’t use \lstalloverstyle yet!
\lst@ifletter
\lst@Output
\lst@ifignore\else
\lst@AppendOther\lst@outputspace
\fi
\else \lst@ifkeepspaces
\lst@AppendOther\lst@outputspace
\else \ifnum\lst@newlines=\z@
\lst@AppendSpecialSpace
\else \ifnum\lst@length=\z@
\global\advance\lst@lostspace\lst@width
\global\advance\lst@pos\m@ne
\else
\lst@AppendSpecialSpace
\fi \fi \fi
\fi \fi}
\lst@whitespacetrue}

\lst@fmtIdentifier

We install a (keyword) test for the ‘format identifiers’.
\lst@InstallTest{f}
\lst@fmt{\lst@gfmt}
\lst@gfmt{wd}
\gdef\lst@fmt@list{\lst@fmt@list}
\gdef\lst@gfmt@list{\lst@gfmt@list}
\gdef\lst@fmtIdentifier#1#2#3#4{\lst@DefOther\lst@fmta{#2}\edef\lst@fmt{\lst@fmt,\lst@fmta}}

\lst@fmt\langle identifier \rangle expands to a \lst@fmtPre/\lst@fmtPost sequence defined by \#2 and \#3.

\lst@EndAspect

\lst@BeginAspect{labels}

\lst@Key{numbers}{none}{%}

\lst@ifletter
\lst@Output
\lst@ifignore\else
\lst@AppendOther\lst@outputspace
\fi
\else \lst@ifkeepspaces
\lst@AppendOther\lst@outputspace
\else \ifnum\lst@newlines=\z@
\lst@AppendSpecialSpace
\else \ifnum\lst@length=\z@
\global\advance\lst@lostspace\lst@width
\global\advance\lst@pos\m@ne
\else
\lst@AppendSpecialSpace
\fi \fi \fi
\fi \fi}
\lst@whitespacetrue}

Formatting identifiers

\lst@fmtIdentifier

We install a (keyword) test for the ‘format identifiers’.
\lst@InstallTest{f}
\lst@fmt{\lst@gfmt}
\lst@gfmt{wd}
\gdef\lst@fmt@list{\lst@fmt@list}
\gdef\lst@gfmt@list{\lst@gfmt@list}
\gdef\lst@fmtIdentifier#1#2#3#4{\lst@DefOther\lst@fmta{#2}\edef\lst@fmt{\lst@fmt,\lst@fmta}}

\lst@fmt\langle identifier \rangle expands to a \lst@fmtPre/\lst@fmtPost sequence defined by \#2 and \#3.

\lst@EndAspect

17.3 Line numbers

Rolf Niepraschk asked for line numbers.
\lst@BeginAspect{labels}

\lst@Key{numbers}{none}{%

167

numbers Depending on the argument we define \lst@PlaceNumber to print the line number.
\lst@Key{numbers}{none}{%
numberstyle  Definition of the keys.
numeralsep  \lst@Key{numberstyle}{}{\def\lst@numberstyle{#1}}
stepnumber  \lst@Key{numeralsep}{10pt}{\def\lst@numeralsep{#1}}
numberblanklines  \lst@Key{stepnumber}{1}{\def\lst@stepnumber{#1\relax}}
numberfirstline  \lst@AddToHook{EmptyStyle}{\let\lst@stepnumber@ne}

firstnumber  We select the first number according to the argument.

\lst@Key{firstnumber}{auto}{\lstKV@SwitchCases{#1}{auto:\let\lst@firstnumber@undefined\%}{last:\let\lst@firstnumber@c@lstnumber}}
\lst@AddToHook{PreSet}{\let\lst@advancenumber\z@}

\lst@firstnumber now set to \lst@lineno instead of \lst@firstline, as per changes in lstpatch.sty from 1.3b pertaining to linerange markers.
\lst@AddToHook{PreInit}{\ifx\lst@firstnumber@undefined\def\lst@firstnumber{\lst@lineno}\fi}

Boris Veytsman proposed to continue line numbers according to listing names. We define the label number of the first printing line here. A bug reported by Jens Schwarzer has been removed by replacing \@ne by \lst@firstline.
\gdef\lst@SetFirstNumber{%\ifx\lst@firstnumber@undefined\@tempcnta 0\csname\@lst no@\lst@intname\endcsname\relax\ifnum\@tempcnta=\z@\@tempcnta\lst@firstnumber@c@lstnumber\else \lst@nololtrue \fi\advance\@tempcnta\lst@advancenumber\edef\lst@firstnumber{\the\@tempcnta}\relax\fi}
\lst@SaveFirstNumber

The current label is stored in \lstno@⟨name⟩. If the name is empty, we use a space instead, which leaves \lstno@@undefined.
\gdef\lst@SaveFirstNumber{%\expandafter\xdef\csname\@lst no@\lst@intname\@empty \else \@lst@intname\fi\endcsname{\the\c@lstnumber}
This counter keeps the current label number. We use it as current label to make line numbers referenced by \ref. This was proposed by Boris Veytsman. We now use \refstepcounter to do the job—thanks to a bug report from Christian Gudrian.

Note that the counter advances before the label is printed and not afterwards. Otherwise we have wrong references—reported by Gregory Van Vooren.

The label number is initialized and we ensure correct line numbers for continued listings. An apparently-extraneous advancement of the line number by \m@ne when firstnumber=last is specified was removed, following a bug report by Joachim Breitner.

Walter E. Brown reported problems with pdftex and hyperref. A bad default of \theHlstlabel was the reason. Heiko Oberdiek found another bug which was due to the localization of \lst@neglisting. He also provided the following fix, replacing \theHlstlisting with the \ifx...\fi construction. Ivo Pletikosić reported another problem with the redefinition of \theHlstlisting. Heiko Oberdiek again provided a fix: \theHlstlisting must be replaced by \theHlstlisting.

There are more things to do. We calculate how many lines must skip their label. The formula is
\[ \text{\lst@skipnumbers} = \text{first printing line} \mod \text{\lst@stepnumber}. \]

Note that we use a nonpositive representative for \lst@skipnumbers.

\begin{verbatim}
\newcounter{lstnumber} \global\c@lstnumber\@ne \init
\renewcommand*{\theHlstlabel}{\@arabic\c@lstnumber}
\lst@AddToHook{EveryPar}{\global\advance\c@lstnumber\lst@advancelstnum
\global\advance\c@lstnumber\m@ne \refstepcounter{lstnumber}\
\lst@SkipOrPrintLabel}\global\let\lst@advancelstnum\@ne

\lst@AddToHook{Init}{\def\@currentlabel{\thelstnumber}}
The label number is initialized and we ensure correct line numbers for
continued listings. An apparently-extraneous advancement of the line number by
\m@ne when firstnumber=last is specified was removed, following a bug report by Joachim Breitner.

\lst@AddToHook{InitVars}{\global\c@lstnumber\lst@firstnumber
\global\advance\c@lstnumber\lst@advancenumber
\global\advance\c@lstnumber\m@ne \refstepcounter{lstnumber}
\global\let\lst@advancelstnum\@ne}

\lst@AddToHook{ExitVars}{\global\advance\c@lstnumber\lst@advancelstnum}

Walter E. Brown reported problems with pdftex and hyperref. A bad default of
\theHlstlabel was the reason. Heiko Oberdiek found another bug which was
due to the localization of \lst@neglisting. He also provided the following fix,
replacing \theHlstlisting with the \ifx...\fi construction. Ivo Pletikosić reported
another problem with the redefinition of \theHlstlisting. Heiko Oberdiek again
provided a fix: \theHlstlisting must be replaced by \theHlstlisting.

\AtBeginDocument{%
\def\theHlstnumber{\ifx\lst@@caption\@empty \lst@neglisting
\else \theHlstlisting \fi
\theHlstnumber}}

\lst@skipnumbers
\end{verbatim}
If \texttt{\lst@stepnumber} is zero, no line numbers are printed:

\begin{verbatim}
\lst@SkipOrPrintLabel
\end{verbatim}

But default is this. We use the fact that \texttt{\lst@skipnumbers} is nonpositive. The counter advances every line and if that counter is zero, we print a line number and decrement the counter by \texttt{\lst@stepnumber}.

\begin{verbatim}
\def\lst@SkipOrPrintLabel{\ifnum\lst@skipnumbers=\z@
\global\advance\lst@skipnumbers-\lst@stepnumber\relax\lst@PlaceNumber\lst@numberfirstlinefalse\else\fi}
\end{verbatim}

If the first line of a listing should get a number, it gets it here.

\begin{verbatim}
\lst@ifnumberfirstline\lst@PlaceNumber\lst@numberfirstlinefalse\fi\global\advance\lst@skipnumbers\@ne\fi
\end{verbatim}

\begin{verbatim}
\def\lst@parshape{\parshape\@ne \z@ \linewidth}
\end{verbatim}

\section*{17.4 Line shape and line breaking

\begin{verbatim}
\def\lst@parshape{\parshape\@ne \z@ \linewidth}
\end{verbatim}

Our first aspect in this section.
xleftmargin Usual stuff.
xrightmargin \lst@Key{xleftmargin}{\z@}{\def\lst@xleftmargin{#1}}
resetmargins \lst@Key{xrightmargin}{\z@}{\def\lst@xrightmargin{#1}}
linewidth \lst@Key{resetmargins}{false}{t}{\lstKV@SetIf{#1}{\lst@ifresetmargins}

The margins become zero if we make an exact box around the listing.
\lst@AddToHook{BoxUnsafe}{\let\lst@xleftmargin\z@}
\let\lst@xrightmargin\z@
\lst@AddToHook{TextStyle}{\let\lst@xleftmargin\z@ \let\lst@xrightmargin\z@ \let\lst@ifresetmargins\iftrue}

Added above hook after bug report from Magnus Lewis-Smith and José Romildo Malaquias respectively.

\lst@parshape The definition itself is easy.
\gdef\lst@parshape{\parshape@ne \@totalleftmargin \linewidth}

We calculate the line width and (inner/outer) indent for a listing.
\lst@AddToHook{Init}\
\lst@ifresetmargins\
\advance\linewidth\@totalleftmargin\relax\
\advance\linewidth\rightmargin\relax\
\@totalleftmargin\z@\
\fi\
\advance\linewidth-\lst@xleftmargin\relax\
\advance\linewidth-\lst@xrightmargin\relax\
\advance\@totalleftmargin\lst@xleftmargin\relax

lineskip The introduction of this key is due to communication with Andreas Bartelt. Version 1.0 implements this feature by redefining \baselinestretch.
\lst@Key{lineskip}{\z@}{\def\lst@lineskip{#1\relax}}
\lst@AddToHook{Init}\
\lst@iflineskip\
\ifdim\z@=\lst@lineskip\
\@tempdima\baselineskip\
\advance\@tempdima\lst@lineskip\
\divide\@tempdima\strip@pt\baselineskip\relax\
multiply\@tempdima\@cclvi\
\divide\@tempdima\baselineskip\relax\
multiply\@tempdima\@cclvi\
\edef\baselinestretch{\strip@pt\@tempdima}\relax\
\selectfont\
\fi\

breaklines As usual we have no problems in announcing more keys. breakatwhitespace is due to Javier Bezos. Unfortunately a previous definition of that key was wrong as Franz Rinnerthaler and Ulrike Fischer reported.
\lst@Key{breaklines}{false}{t}{\lstKV@SetIf{#1}{\lst@ifbreaklines}}
We assign some different macros and (if necessary) suppress “underfull \hbox” messages (and use different pretolerance):

```
\lst@AddToHook{Init}{\lst@ifbreaklines
\badness\@M \pretolerance\@M
\rightskip\@flushglue \rightskip \rightskip % \raggedright
\leftskip\z@skip \parindent\z@
\ifbreakautoindent \ifwhitespace \lst@@discretionary \fi\else \lst@@discretionary \fi}
```

A \raggedright above has been replaced by setting the values by hand after a bug report from Morten Høgholm.

We use the normal parshape and the calculated \lst@breakshape (see below).

```
\def\lst@parshape{\parshape\tw@ \@totalleftmargin \linewidth
\lst@breakshape}\else \let\lst@discretionary\@empty \fi}
```

```
\lst@AddToHook{OnNewLine}{\lst@ifbreaklines \lst@breakNewLine \fi}
```

Here comes the whole magic: We set a discretionary break after each ‘output unit’. However we redefine \space to be used inside \discretionary and use EveryLine hook. After a bug report by Carsten Hamm I’ve added \kern-\lst@xleftmargin, which became \kern-\@totalleftmargin after a bug report by Christian Kaiser.

```
\gdef\lst@discretionary{%
\ifbreakautoindent \ifwhitespace \lst@@discretionary \fi\else \lst@@discretionary \fi}
```

```
\lst@AddToHook{PostOutput}{\lst@discretionary}
```

Alternative: \penalty\@M \hskip\z@ plus 1fil \penalty0 \hskip\z@ plus-1fil before each ‘output unit’ (i.e. before \hbox{...} in the output macros) also break the lines as desired. But we wouldn’t have prebreak and postbreak.

We use breakindent, and additionally the current line indentation (coming from white spaces at the beginning of the line) if ‘auto indent’ is on.

```
\gdef\lst@breakNewLine{%
\tempdima\lst@breakindent
\ifbreakautoindent \advance\tempdima\lostspace \fi
```

172
Now we calculate the margin and line width of the wrapped part...

\@tempdimc-\@tempdima \advance\@tempdimc\linewidth
\advance\@tempdima\@totalleftmargin

... and store it in \lst@breakshape.
\xdef\lst@breakshape{\noexpand\lst@breakcurrindent \the\@tempdimc}
\xdef\lst@breakcurrindent{\the\@tempdima}
\global\let\lst@breakcurrindent\z@ % init

The initialization of \lst@breakcurrindent has been added after a bug report by Alvaro Herrera.

To do: We could speed this up by allocating two global dimensions.

\lst@breakshape
Andreas Deininger reported a problem which is resolved by providing a default break shape.
\gdef\lst@breakshape{\@totalleftmargin \linewidth\
\lst@breakProcessOther is the same as \lst@ProcessOther except that it also outputs the current token string. This inserts a potential linebreak point. Only the closing parenthesis uses this macro yet.
\gdef\lst@breakProcessOther#1{\lst@ProcessOther#1\lst@OutputOther}
\lst@AddToHook{SelectCharTable}{\lst@ifbreaklines \lst@Def{)}{\lst@breakProcessOther)}\fi}

A bug reported by Gabriel Tauro has been removed by using \lst@ProcessOther instead of \lst@AppendOther.
\lst@EndAspect
\lst@BeginAspect\[lineshape\]{frames}

framexleftmargin These keys just save the argument.
framexrightmargin \lst@Key{framexleftmargin}{\z@}{\def\lst@framexleftmargin{#1}}
framextopmargin \lst@Key{framexrightmargin}{\z@}{\def\lst@framextopmargin{#1}}
framexbottommargin \lst@Key{framexleftmargin}{\z@}{\def\lst@framexbottommargin{#1}}
framexbottommargin

backgroundcolor Ralf Imh"auser inspired the key backgroundcolor. All keys save the argument, and...
\lst@Key{backgroundcolor}{\z@}{\def\lst@bkgcolor{#1}}
\lst@Key{fillcolor}{\z@}{\def\lst@fillcolor{#1}}
\lst@Key{rulecolor}{\z@}{\def\lst@rulecolor{#1}}
\lst@Key{rulesepcolor}{\z@}{\def\lst@rulesepcolor{#1}}

... some have default settings if they are empty.
\lst@AddToHook{Init}{\ifx\lst@fillcolor\@empty \let\lst@fillcolor\lst@bkgcolor \fi}
2832 \ifx\lst@rulesepcolor\@empty
2833 \let\lst@rulesepcolor\lst@fillcolor
2834 \fi

\begin{tabular}{l}
\textbf{rulesep} Another set of keys, which mainly save their respective argument. \textbf{frameshape}
\end{tabular}

\begin{tabular}{l}
\textbf{framerule} capitalizes all letters, and checks whether at least one round corner is specified.
\end{tabular}

\begin{tabular}{l}
\textbf{framesep} Eventually we define \texttt{\lst@frame} to be empty if and only if there is no frameshape.
\end{tabular}

\begin{tabular}{l}
\texttt{\lst@Key{rulesep}{2pt}{\def\lst@rulesep{#1}}}\par
\texttt{\lst@Key{framerule}{.4pt}{\def\lst@framerulewidth{#1}}}\par
\texttt{\lst@Key{framesep}{3pt}{\def\lst@frametextsep{#1}}}\par
\texttt{\lst@Key{frameshape}{}{}}\par
\texttt{\let\lst@xrulecolor\@empty\par}
\texttt{\lstKV@FourArg{#1}{\uppercase\def\lst@frametshape{##1}}{\uppercase\def\lst@framelshape{##2}}{\uppercase\def\lst@framershape{##3}}{\uppercase\def\lst@framebshape{##4}}\par}
\texttt{\let\lst@ifframeround\iffalse\par}
\texttt{\lst@IfSubstring R\lst@frametshape\let\lst@ifframeround\iftrue\fi\par}
\texttt{\lst@IfSubstring R\lst@framebshape\let\lst@ifframeround\iftrue\fi\par}
\texttt{\def\lst@frame{##1##2##3##4}}\par
\end{tabular}

\begin{tabular}{l}
\textbf{frameround} We have to do some conversion here.
\end{tabular}

\begin{tabular}{l}
\texttt{\lst@Key{frameround}\relax\par}
\texttt{\uppercase\def\lst@frameround{#1}}\par
\texttt{\expandafter\lstframe@\lst@frameround ffff\relax}\par
\end{tabular}

\begin{tabular}{l}
In case of an verbose argument, we use the \texttt{trbl}-subset replacement.
\end{tabular}

\begin{tabular}{l}
\texttt{\lst@Key{frame}\relax\par}
\texttt{\let\lst@xrulecolor\@empty\par}
\texttt{\lstKV@SwitchCases{#1}{none:\let\lst@frame\@empty\par}
\texttt{leftline:\def\lst@frame{l}\par}
\texttt{topline:\def\lst@frame{t}\par}
\texttt{bottomline:\def\lst@frame(b)\par}
\texttt{lines:\def\lst@frame(tb)\par}
\texttt{single:\def\lst@frame(trbl)\par}
\texttt{shadowbox:\def\lst@frame(tRBl)\par}
\texttt{\def\lst@xrulecolor{\lst@rulesepcolor}\par}
\texttt{\def\lst@rulesep{\lst@frametextsep}\par}
\texttt{\def\lst@frametshape{\lst@frametshape}\par}
\texttt{\def\lst@framelshape{\lst@framelshape}\par}
\texttt{\def\lst@framershape{\lst@framershape}\par}
\texttt{\def\lst@framebshape{\lst@framebshape}\par}
\texttt{}}\par
\texttt{\expandafter\lstframe@\lst@frameround ffff\relax}\par
\end{tabular}

\begin{tabular}{l}
Adding \texttt{t}, \texttt{r}, \texttt{b}, and \texttt{1} in case of their upper case versions makes later tests easier.
\end{tabular}

\begin{tabular}{l}
\texttt{\lstframe@#1#2#3#4#5}\par
\texttt{\lst@IfSubstring T\lst@framelshape\let\lst@frametshape{\lst@frametshape}\par}
\texttt{\lst@IfSubstring R\lst@frametshape\let\lst@frametshape{\lst@frametshape}\par}
\texttt{\lst@IfSubstring B\lst@frametshape\let\lst@frametshape{\lst@frametshape}\par}
\texttt{\lst@IfSubstring L\lst@frametshape\let\lst@frametshape{\lst@frametshape}\par}
\texttt{}}\par
\texttt{\expandafter\lstframe@\lst@frameround ffff\relax}\par
\end{tabular}

\begin{tabular}{l}
We now check top and bottom frame rules, . . .
\end{tabular}

\begin{tabular}{l}
\texttt{\lst@frametshape\empty\let\lst@frametshape\empty\par}
\texttt{\lst@frametCheck\par}
\texttt{ltr\lst@frametshape\lst@frametshape\lst@frametshape #4#1}\par
\end{tabular}
\lst@frameCheck
  LTR\lst@framelshape\lst@frametshape\lst@framershape #4\#1%
\lst@frameCheck
  lbr\lst@framelshape\lst@frametshape\lst@framershape #3\#2%
\lst@frameCheck
  LBR\lst@framelshape\lst@frametshape\lst@framershape #3\#2%

... look for round corners ... 
\let\lst@ifframeround\iffalse
\lst@IfSubstring R\lst@frametshape{\let\lst@ifframeround\iftrue}{}%
\lst@IfSubstring R\lst@framebshape{\let\lst@ifframeround\iftrue}{}%

and define left and right frame shape.
\let\lst@framelshape\@empty \let\lst@framershape\@empty
\lst@IfSubstring L\lst@frame{
  \def\lst@framelshape{YY}}%
\lst@IfSubstring l\lst@frame{\def\lst@framelshape{Y}}{}
\lst@IfSubstring R\lst@frame{
  \def\lst@framershape{YY}}%
\lst@IfSubstring r\lst@frame{\def\lst@framershape{Y}}{}

Now comes the macro used to define top and bottom frame shape. It extends the
macro #5. The last two arguments show whether left and right corners are round.
#4 and #6 are temporary macros. #1#2#3 are the three characters we test for.
\gdef\lst@frameCheck#1#2#3#4#5#6#7#8{%
  \lst@IfSubstring #1\lst@frame{
    \if #7T\def#4{R}\else \def#4{Y}\fi}%
  \def#4{N}%
  \lst@IfSubstring #3\lst@frame{
    \if #8T\def#6{R}\else \def#6{Y}\fi}%
  \def#6{N}%
  \lst@IfSubstring #2\lst@frame{
    \edef#5{#5#4Y#6}}{}
}\lst@frameCheck

For text style listings all frames and the background color are deactivated – added
after bug reports by Stephen Reindl and Thomas ten Cate
\lst@AddToHook{TextStyle}{
  \let\lst@frame\@empty
  \let\lst@frametshape\@empty
  \let\lst@framershape\@empty
  \let\lst@framebshape\@empty
  \let\lst@framelshape\@empty
  \let\lst@bkgcolor\@empty}

As per a bug report by Ignacio Fernández Galván, the small section of background
color to the left of the margin is now drawn before the left side of the
frame is drawn, so that they overlap correctly in Acrobat.
\kern\lst@framexleftmargin}\%
\fi
\llap{\setbox\z@\hbox{\vrule\@width\z@\@height#2\@depth#3\%
\lst@frameL}\%
\rlap{\lst@frameBlock\lst@rulesepcolor{\wd\z@}\%
{\ht\z@}{\dp\z@}}\%}
\box\z@
\kern\lst@frametextsep\relax
\kern\lst@framexleftmargin}\%
\rlap{\kern-\lst@framexleftmargin
\@tempdima\linewidth
\advance\@tempdima\lst@framexleftmargin
\advance\@tempdima\lst@framexrightmargin
\lst@frameBlock\lst@bkgcolor\@tempdima(#2){#3}\%
\ifx\lst@framershape\@empty
\kern\lst@frametextsep\relax
\else
\lst@frameBlock\lst@fillcolor\lst@frametextsep(#2){#3}\%
\fi
\setbox\z@\hbox{\vrule\@height\z@\@depth#3\%
\lst@frameR}\%
\rlap{\lst@frameBlock\lst@rulesepcolor{\wd\z@}\%
{\ht\z@}{\dp\z@}}\%
\box\z@\%
\color@endgroup})

\lst@frametable
\gdef\lst@frametable\#3#4{%
\color@begingroup
#3%
\setbox\z@\hbox{\vrule\@height#3\@depth#4\%
\ifx#3\@empty \@width\z@ \kern#2\relax
\else \@width#2\relax \fi}\%
\box\z@
\color@endgroup}

\lst@frameR \typesets right rules. We only need to iterate through \lst@framershape.
\gdef\lst@frameR{%
\expandafter{\lst@frameR\@}\lst@framershape\relax
\kern-\lst@rulesep}
\gdef\lst@frameR#1{%
\if\relax#1\@empty else
\kern\lst@frametable\if #1Y\lst@framevrule \else \kern\lst@framerulewidth \fi
\kern\lst@rulesep
\expandafter{\lst@frameRb
\fi}
\gdef\lst@frameRb#1{%
\if\relax#1\@empty\else
\kern#2
\lst@xrulecolor
\lst@framevrule
\color@endgroup
\else
\if #1Y\color@begingroup
\lst@xrulecolor
\lst@framevrule
\color@endgroup
\else
\end{document}
Ditto left rules.

\lst@frameL Ditto left rules.
\lst@frameH This is the central macro used to draw top and bottom frame rules. The first argument is either T or B and the second contains the shape. We use \@tempcntb as size counter.

\lst@frameH#1#2{\global\let\lst@framediml\z@ \global\let\lst@framedimr\z@ \setbox\z@\hbox{}\@tempcntb\z@ \expandafter\lst@frameH0\expandafter\#1\empty\relax\relax \@tempdimb\lst@frametextsep\relax \advance\@tempdimb\lst@framerulewidth\relax \@tempdimc\ht\z@ \advance\@tempdimc\dp\z@ \setbox\z@=\hbox{\lst@frameHBkg\lst@fillcolor\@tempdimb\@firstoftwo \if#1T\rlap{\raise\dp\@tempboxa\box\@tempboxa}\fi \lst@frameHBkg\lst@rulesepcolor\@tempdimc\@secondoftwo \advance\@tempdimb\ht\@tempboxa \if#1T\rlap{\raise\lst@frametextsep\box\@tempboxa}\fi \else\rlap{\lower\@tempdimb\box\@tempboxa}\fi \rlap{\box\z@}}\@tempcntb=\z@}
\lst@frameh This is the low-level macro used to draw top and bottom frame rules. It adds one rule plus corners to box 0. The first parameter gives the size of the corners and the second is either T or B. \#3\#4\#5 is a left-to-right description of the frame and is in \{Y,N,R\} × \{Y,N\} × \{Y,N,R\}. We move to the correct horizontal position, set the left corner, the horizontal line, and the right corner.

\lst@frameh\#1\#2\#3\#4\#5{\kern\lst@framexleftmargin\color@begingroup\ifnum\#1=\z@ \lst@rulecolor \else \lst@xrulecolor \fi
\lst@frameCornerX\llap{#2L}#3\#1\ifdim\lst@framediml<\@tempdimb\xdef\lst@framediml{\the\@tempdimb}\fi\begingroup\if#4Y\else \let\lst@framerulewidth\z@ \fi\vrule\@width\@tempdima\@height\lst@framerulewidth \@depth\z@\endgroup
\lst@frameCornerX\rlap{#2R}#5\#1\ifdim\lst@framedimr<\@tempdimb\xdef\lst@framedimr{\the\@tempdimb}\fi}}

\lst@frameCorner gets four arguments: \llap, TL or BL, the corner type \in\{Y,N,R\}, and the size \#1.

\lst@frameCornerX\llap{#2L}\#3\#1\ifdim\lst@framediml<\@tempdima\xdef\lst@framediml{\the\@tempdima}\fi
\begin{group}\if#4Y\else \let\lst@framerulewidth\z@ \fi\vrule\@width\@tempdima\@height\lst@framerulewidth \@depth\z@\endgroup
\lst@frameCornerX\rlap{#2R}\#5\#1\ifdim\lst@framediml<\@tempdima\xdef\lst@framediml{\the\@tempdima}\fi
\lst@frameCornerX\ typesets a single corner and returns $\@tempdimb$, the width of the corner.

\lst@frameCalcDimA\ calculates an all over width; used by \lst@frameh\ and \lst@frameInit.\n
\lst@frameInit\ First we look which frame types we have on the left and on the right. We speed up things if there are no vertical rules.

\lst@ifframeround\ We adjust values to round corners if necessary.

179
Finally we typeset the rules (+ corners). We possibly need to insert negative \vskip to remove space between preceding text and top rule.

To do: Use \vspace instead of \vskip?

\lst@frametshape\empty\else
\lst@frameH T\lst@frametshape
\ifvoid\z@\else

\par\lst@parshape
\@tempdima-\baselineskip \advance\@tempdima\ht\z@  
\ifdim\prevdepth<\@cclvi\p@\else
\advance\@tempdima\prevdepth
\fi
\fi
\ifdim\@tempdima<\z@
\vskip\@tempdima\vskip\lineskip
\fi
\noindent\box\z@\par
\lineskiplimit\maxdimen \lineskip\z@
\fi
\lst@frameSpreadV\lst@framextopmargin
\fi}
\parshape\lst@parshape
ensures that the top rules correctly indented. The bug was reported by Marcin Kasperski.

We typeset left and right rules every line.
\lst@AddToHook{EveryLine}{\lst@framelr}
\global\let\lst@framelr\@empty
\lst@frameExit

The rules at the bottom.
\lst@AddToHook{DeInit}{\ifx\lst@framebshape\@empty\else \lst@frameExit \fi}
\gdef\lst@frameExit{%
\lst@frameSpreadV\lst@framexbottommargin
\lst@frameH B\lst@frametshape
\ifvoid\z@\else
\everypar{}\par\lst@parshape\nointerlineskip\noindent\box\z@
\fi}
\lst@frameSpreadV sets rules for vertical spread.
\gdef\lst@frameSpreadV#1{%
\ifdim\z@=#1\else
\everypar{}\par\lst@parshape\nointerlineskip\noindent
\lst@frameMakeBoxV\z@{#1}{\z@}{%}
\box\z@
\fi}
\lst@frameTR These macros make a vertical and horizontal rule. The implicit argument \@tempdima gives the size of two corners and is provided by \lst@frameh.
\lst@frameBL\lst@frameTL
\gdef\lst@frameTR{%
\vrule\@width.5\@tempdima\@height\lst@framerulewidth\@depth\z@
\kern-\lst@framerulewidth
\raise\lst@framerulewidth\bbox{%
\vrule\@width\lst@framerulewidth\@height\z@\@depth.5\@tempdima}}
\gdef\lst@frameBL{%
\lst@frameRoundT are helper macros to typeset round corners. We set height and depth to the visible parts of the circle font.

\lst@frameRoundB

\lst@frameRTR
\lst@frameRBR
\lst@frameRBL
\lst@frameRTL

The round corners.

\lst@frameRTR
\lst@frameRBR
\lst@frameRBL
\lst@frameRTL

17.6 Macro use for make

If we've entered the special mode for Make, we save whether the last identifier has been a first order keyword.
makemacrouse adjusts the character table if necessary

\lst@MakeSCT If `macro use` is on, the opening \$( prints preceding characters, enters the special mode and merges the two characters with the following output.

\lst@lineno print firstline lastline linerange

\lst@lineno \newcount\lst@lineno % \global
\lst@AddToHook{InitVars}{\global\lst@lineno\@ne}
\lst@Key{print}{true}\[t\]{\lstKV@SetIf{#1}\lst@ifprint}
\lst@Key{firstline}\relax{\def\lst@firstline{#1\relax}}
\lst@Key{lastline}\relax{\def\lst@lastline{#1\relax}}
\lst@AddToHook{PreSet}{\let\lst@firstline\@ne \def\lst@lastline{9999999\relax}}
\lst@EndAspect

\lst@lineno The ‘current line’ counter and three keys.

\lst@lineno \newcount\lst@lineno % \global
\lst@AddToHook{InitVars}{\global\lst@lineno\@ne}
\lst@Key{print}{true}\[t\]{\lstKV@SetIf{#1}\lst@ifprint}
\lst@Key{firstline}\relax{\def\lst@firstline{#1\relax}}
\lst@Key{lastline}\relax{\def\lst@lastline{#1\relax}}
\lst@AddToHook{PreSet}{\let\lst@firstline\@ne \def\lst@lastline{9999999\relax}}
The following code is just copied from the current development version, and from the `lstpatch.sty` file that Carsten left in version 1.3b for doing line ranges with numbers and range markers. It uses some more keys:

- `linerange` First, the options that control the line-range handling.

- `rangeprefix` 3216 \lst@Key{linerange}\relax\{\lstKV@OptArg[^{\#1}]{{#1}}\}

- `rangesuffix` 3217 \def\lst@interrange{\#1}\def\lst@linerange{\#2,}

- `rangebeginprefix` 3218 \lst@Key{rangeprefix}\relax\{\def\lst@rangebeginprefix{\#1}\}

- `rangeendsuffix` 3219 \def\lst@rangeendprefix{\#1}

- `rangesuffix` 3220 \lst@Key{rangesuffix}\relax\{\def\lst@rangeendsuffix{\#1}\}

- `rangebeginprefix` 3221 \lst@Key{rangebeginprefix}{}{\def\lst@rangebeginprefix{\#1}}

- `rangebeginsuffix` 3222 \lst@Key{rangebeginsuffix}{}{\def\lst@rangebeginsuffix{\#1}}

- `rangeendprefix` 3223 \lst@Key{rangeendprefix}{}{\def\lst@rangeendprefix{\#1}}

- `rangeendsuffix` 3224 \lst@Key{rangeendsuffix}{}{\def\lst@rangeendsuffix{\#1}}

- `includerangemarker` 3225 \lst@Key{includerangemarker}{true}\{t\}{\lstKV@SetIf{\#1}\lst@ifincluderangemarker}

If we've found a general marker, we set firstline and lastline to 9999999. This prevents (almost) anything from being printed for now.

183
The following definition will be executed in the \SelectCharTable hook and here right now if we are already processing a listing.

\GlobalDef\lst@DefRange{\expandafter\lst@CArgX\lst@rangebegin\relax\lst@DefRangeB}
\ifnum\lst@mode=\lst@Pmode \expandafter\lst@DefRange \fi}

\lst@DefRange is not inserted via a hook anymore. Instead it is now called directly from \lst@SelectCharTable. This was necessary to get rid of an interference with the escape-to-LaTeX-feature. The bug was reported by Michael Bachmann.

Another chance is due to the same bug: \lst@DefRange is redefined globally when the begin of code is found, see below. The bug was reported by Tobias Rapp and Markus Luisser.

\lst@AddToHookExe{DeInit}{\global\let\lst@DefRange\@empty}

Actually defining the marker (via \lst@CCLI, \lst@DefRange, \lst@CArgX as seen above) is similar to \lst@DefDelimB—except that we unfold the first parameter and use different \execute, \pre, and \post statements.

\def\lst@DefRangeB#1#2\def\lst@DefRangeB@#1#2#3#4{\lst@CDef{#1{#2}{#3}}#4{}\if\ifincluderangemarker\lst@LeaveMode\let#1#4\lst@DefRangeEnd\lst@InitLstNumber\else\@tempcnta\lst@lineno \advance\@tempcnta\@ne\edef\lst@firstline{\the\@tempcnta}\gdef\lst@OnceAtEOL{\let#1\lst@DefRangeEnd}\lst@InitLstNumber\fi\global\let\lst@DefRange\lst@DefRangeEnd\lst@CArgEmpty}}

Modify labels and define \lst@InitLstNumber used above according to an error reported by Omair-Inam Abdul-Matin.

\def\lstpatch@labels{%
\gdef\lst@SetFirstNumber{%\ifx\lst@firstnumber\@undefined\@tempcnta 0\csname\@lst no@\lst@intname\endcsname\relax\edef\lst@firstline{\the\@tempcnta}\relax\else\lst@nololtrue\advance\@tempcnta\lst@advancenumber\edef\lst@firstline{\the\@tempcnta}\relax\else\lst@nololtrue\advance\@tempcnta\lst@advancenumber\edef\lst@firstline{\the\@tempcnta}\relax\fi\fi}%

\global\let\lst@DefRange\lst@DefRangeEnd\lst@CArgEmpty}%

\@empty}

Modifying labels and define \lst@InitLstNumber used above according to an error reported by Omair-Inam Abdul-Matin.
Byron K. Boulton reported, that the line numbers are off by one, if they are displayed when a linelrange is given by patterns and includerangemarker=false is set. Adding this test corrects this behaviour.

The end-marker is defined if and only if it’s not empty. The definition is similar to \lst@DefDelimE—with the above exceptions and except that we define the re-entry point \lst@DefRangeE@@ as it is defined in the new version of \lst@MProcessListing above.

Eventually we shouldn’t forget to install \lst@OnceAtEOL, which must also be called in \lst@MSkipToFirst.
Finally the service macro \lst@IfNumber:

\def\lst@IfNumber#1{\ifx\@empty#1\@empty\let\lst@next\@firstoftwo\else\lst@IfNumber@#1\@nil\fi}
\def\lst@IfNumber@#1#2\@nil{\let\lst@next\@secondoftwo\ifnum'#1>47\relax\ifnum'#1>57\relax\let\lst@next\@firstoftwo\fi\fi}

nolol is just a key here. We’ll use it below, of course.
\lst@Key{nolol}{false}[t]{\lstKV@SetIf{#1}\lst@ifnolol}
\def\lst@nololtrue{\let\lst@ifnolol\iftrue}
\let\lst@ifnolol\iffalse % init

\section{Floats, boxes and captions}

Some keys and ... captionpos
\lst@Key{captionpos}{t}{\def\lst@captionpos{#1}}
\lst@Key{abovecaptionskip}{\smallskipamount}{\def\lst@abovecaption{#1}}
\lst@Key{belowcaptionskip}{\smallskipamount}{\def\lst@belowcaption{#1}}
\lst@Key{label}{\relax}{\def\lst@label{#1}}
\lst@Key{title}{\relax}{\def\lst@title{#1}\let\lst@caption\relax}
\lst@Key{caption}{\relax}{\lstKV@OptArg\[{#1}\]{#1}\def\lst@caption{##2}\def\lst@@caption{##1}}
\let\lst@title\@empty
\lst@AddToHookExe{TextStyle}{\let\lst@caption\@empty \let\lst@@caption\@empty \let\lst@title\@empty \let\lst@label\@empty

\thelstlisting
... and how the caption numbers look like. I switched to \@ifundefined (instead of \ifx \@undefined) after an error report from Denis Girou.

This is set \AtBeginDocument so that the user can specify whether or not the counter should be reset at each chapter before the counter is defined, using the numberbychapter key.
Before defining this macro, we ensure that some other control sequences exist—Adam Prugel-Bennett reported problems with the slides document class. In particular we allocate above- and belowcaption skip registers and define \@makecaption, which is an exact copy of the definition in the article class. To respect the LPPL: you should have a copy of this class on your \TeX system or you can obtain a copy from the CTAN, e.g. from the ftp-server \texttt{ftp.dante.de}.

Axel Sommerfeldt proposed a couple of improvements regarding captions and titles. The first is to separate the definitions of the skip registers and \@makecaption. The introduction of \fnum@lstlisting is also due to Axel. Previously the replacement text was used directly in \lst@MakeCaption. A \texttt{\noindent} has been moved elsewhere and became \texttt{\parboxrestore} after a bug report from Frank Mittelbach. Karl Berry asked for the ability of customizing the label. So \texttt{\lstlistingnamestyle} was introduced in front of \texttt{\lstlistingname}.

The introduction of \fnum@lstlisting is also due to Axel. Previously the replacement text was used directly in \lst@MakeCaption. A \texttt{\noindent} has been moved elsewhere and became \texttt{\parboxrestore} after a bug report from Frank Mittelbach. Karl Berry asked for the ability of customizing the label. So \texttt{\lstlistingnamestyle} was introduced in front of \texttt{\lstlistingname}.

Captions are set only for display style listings – thanks to Peter Löffler for reporting the bug and to Axel Sommerfeldt for analyzing the bug. We \texttt{\refstepcounter} the listing counter if and only if \texttt{\lst@@caption} is not empty. Otherwise we ensure correct hyper-references, see \texttt{\lst@HRefStepCounter} below. We do this once a listing, namely at the top.
The following code has been moved here from the Init hook after a bug report from Rolf Niepraschk. Moreover the initialization of \lst@name et al have been inserted here after a bug report from Werner Struckmann. We make a ‘lol’ entry if the name is neither empty nor a single space. But we test \lst@caption and \lst@ifnolol first.

We make a caption if and only if the caption is not empty and the user requested a caption at $#1 \in \{t, b\}$. To disallow pagebreaks between caption (or title) and a listing, we redefine the primitive \vskip locally to insert \nobreak. Note that we allow pagebreaks in front of a ‘top-caption’ and after a ‘bottom-caption’. Also, the \ignorespaces in the \@makecaption call is added to match what \LaTeX does in \@caption; the AMSSbook class (and perhaps others) assume this is present and attempt to strip it off when testing for an empty caption, causing a bug noted by Xiaobo Peng.

To do: This redefinition is a brute force method. Is there a better one?
I’ve inserted \normalsize after a bug report from Andreas Matthias and moved it front of \@makecaption after receiving another from Sonja Weidmann.

Axel proposed the first definition. The other two are default definitions. They may be adjusted to make listings compatible with other packages and classes.

The following caption(2) support comes also from Axel.

This macro sets the listing number to a negative value since the user shouldn’t refer to such a listing. If the hyperref package is present, we use ‘lstlisting’ (argument from above) to hyperref to. The groups have been added to prevent other packages (namely tabularx) from reading the locally changed counter and writing it back globally. Thanks to Michael Niedermair for the report. Unfortunately this localization led to another bug, see \theHlstnumber.

boxpos sets the vertical alignment of the (possibly) used box respectively indicates that a box is used.

Matthias Zenger asked for double-column floats, so I’ve inserted some code. We first check for a star ...

float placement first check for a star ...
The float type \texttt{float} is set according to whether the \texttt{float} package is loaded and whether figure and table floats are defined. This is done at \texttt{\begin{document}} to make the code independent of the order of package loading.

\begin{verbatim}
\AtBeginDocument{\if\undefined\c@float@type
\edef\ftype@lstlisting{\if\@undefined\c@figure 1\else 4\fi}
\else
\edef\ftype@lstlisting{\the\c@float@type}
\addtocounter{float@type}{\value{float@type}}
\fi}
\end{verbatim}

\section{Init and EOL}

\begin{description}
\item[\texttt{aboveskip}] We define and initialize these keys and prevent extra spacing for `inline` listings (in particular if \texttt{fancyvrb} interface is active, problem reported by Denis Girou).
\item[\texttt{belowskip}] \begin{verbatim}
\lst@Key{aboveskip} \medskipamount {\def\lst@aboveskip{#1}}
\lst@Key{belowskip} \medskipamount {\def\lst@belowskip{#1}}
\end{verbatim}
\item[\texttt{everydisplay}] Some things depend on display-style listings.
\end{description}

\begin{verbatim}
\lst@ifdisplaystyle
\lst@Key{everydisplay} {\def\lst@EveryDisplay{#1}}
\lst@AddToHook{TextStyle}{\let\lst@ifdisplaystyle\iffalse}
\lst@AddToHook{DisplayStyle}{\let\lst@ifdisplaystyle\iftrue}
\let\lst@ifdisplaystyle\iffalse
\fi
\end{verbatim}
In restricted horizontal \TeX mode we switch to \texttt{\lst@boxtrue}. In that case we make appropriate box(es) around the listing.

Some initialization. I removed \texttt{\parskip} after bug report from Jim Hefferon. He reported the same problem as Aidan Philip Heerdegen (see below), but I immediately saw the bug here since Jim used \texttt{\parskip} ≠ 0.
The end of line character chr(13)=^M controls the processing, see the definition of \lst@MProcessListing below. The argument #1 is either \relax or \lstenv@backslash.

Note: From version 0.19 on ‘listing processing’ is implemented as an internal mode, namely a mode with special character table. Since a bug report from Fermin Reig \rightskip and the others are reset via PreInit and not via InitVars.

showlines lets the user control whether empty lines at the end of a listing are printed. But you know that if you’ve read the User’s guide.

\lst@DeInit Output the remaining characters and update all things. First I missed to use \lst@ifdisplaystyle here, but then KP Gores reported a problem. The \everypar has been put behind \lsthk@ExitVars after a bug report by Michael Niedermair and I’ve added \normalbaselines after a bug report by Georg Rehm and \normalcolor after a report by Walter E. Brown.

Place the bottom caption.

Close the boxes if necessary and make a rule to get the right width. I added the \par\nointerlineskip (and removed \nointerlineskip later again) after receiving a bug report from Aidan Philip Heerdegen. \everypar{} is due to a bug report from Sonja Weidmann.
End the multicolumn environment and/or float if necessary.
\ifx\lst@multicols\@empty\else
  \def\lst@next{\global\let\@checkend\@gobble
    \endmulticols
    \global\let\@checkend\lst@@checkend}
\expandafter\lst@next
\fi\ifx\lst@float\relax\else
  \expandafter\lst@endfloat
\fi
\endgroup}
\let\lst@@checkend\@checkend

\lst@maxwidth is to be allocated, initialized and updated.
\newdimen\lst@maxwidth % \global
\lst@AddToHook{InitVars}{\global\lst@maxwidth\z@}
\lst@AddToHook{InitVarsEOL}{\ifdim\lst@currlwidth>\lst@maxwidth
  \global\lst@maxwidth\lst@currlwidth
  \fi}
\lst@EOLUpdate
What do you think this macro does?
\def\lst@EOLUpdate{\lsthk@EOL \lsthk@InitVarsEOL}
\lst@MProcessListing
This is what we have to do at EOL while processing a listing. We output all
remaining characters and update the variables. If we’ve reached the last line, we
check whether there is a next line interval to input or not.
\def\lst@MProcessListing{%
  \lst@XPrintToken \lst@EOLUpdate \lsthk@InitVarsBOL
  \global\advance\lst@lineno\@ne
  \ifnum\lst@lineno>\lst@lastline
    \lst@ifdropinput \lst@LeaveMode \fi
    \ifnum\lst@linenorm>\lst@linenol\@ne
      \lst@ifdropinput \lst@LeaveMode \fi
    \ifx\lst@linerange\@empty
      \expandafter\expandafter\expandafter\lst@EndProcessListing
    \else
      \lst@interrange
      \lst@GetLineInterval
      \expandafter\expandafter\expandafter\lst@SkipToFirst
    \fi
  \else
    \expandafter\lst@BOLGobble
  \fi}
\lst@EndProcessListing
Default definition is \endinput. This works for \lstinputlisting.
\let\lst@EndProcessListing\endinput
\lst@gobble
The key sets the number of characters to gobble each line.
\lst@Key{gobble}{0}{\def\lst@gobble{#1}}
If the number is positive, we set a temporary counter and start a loop.

\texttt{\lst@BOLGobble} \texttt{\def\lst@BOLGobble{\ifnum\lst@gobble>\z@\@tempcnta\lst@gobble\relax\expandafter\lst@BOLGobble@\fi}}

A nonpositive number terminates the loop (by not continuing). Note: This is not the macro just used in \texttt{\lst@BOLGobble}.

\texttt{\def\lst@BOLGobble@@{\ifnum\@tempcnta>\z@\expandafter\lst@BOLGobble@\fi}}

If we gobble a backslash, we have to look whether this backslash ends an environment. Whether the coming characters equal e.g. \texttt{end\{lstlisting\}}, we either end the environment or insert all just eaten characters after the ‘continue loop’ macro.

\texttt{\def\lstenv@BOLGobble@@{\lst@IfNextChars\lstenv@endstring{\lstenv@End}{\advance\@tempcnta\m@ne \expandafter\lst@BOLGobble@@\lst@eaten}}} \texttt{(\lst@IfNextChars\lstenv@endstring{\lstenv@End}{\advance\@tempcnta\m@ne \expandafter\lst@BOLGobble@@\lst@eaten})}

Now comes the loop: if we read \texttt{relax}, EOL or FF, the next operation is exactly the same token. Note that for FF (and tabs below) we test against a macro which contains \texttt{\lst@ProcessFormFeed}. This was a bug analyzed by Heiko Oberdiek.

\texttt{\def\lst@BOLGobble@#1{\let\lst@next#1% \ifx\lst@next\relax\else \ifx\lst@next\lst@MProcessListing\else \ifx\lst@next\lst@processformfeed\else Otherwise we use one of the two submacros.

\texttt{\ifx\lst@next\lstenv@backslash\let\lst@next\lstenv@BOLGobble@@ \else \let\lst@next\lst@BOLGobble@@ \fi}}

Now we really gobble characters. A tabulator decreases the temporary counter by \texttt{\lst@tabsize} (and deals with remaining amounts, if necessary), …

\texttt{\ifx\lst@processtabulator\texttt{(\advance\@tempcnta-\lst@tabsize}\relax \ifnum\@tempcnta<\z@ \texttt{\lst@length-\@tempcnta \lst@PreGotoTabStop}} \texttt{\fi}}

… whereas any other character decreases the counter by one.

\texttt{\else \advance\@tempcnta\m@ne \fi \fi \fi \fi} \texttt{\lst@BOLGobble@@} \texttt{\def\lst@processformfeed{\lst@ProcessFormFeed}} \texttt{\def\lst@processtabulator{\lst@ProcessTabulator}}
18.3 List of listings

name Each pretty-printing command values \lst@intname before setting any keys.
\lstname 3660 \lst@Key(name)\relax{\def\lst@intname{#1}}
\lst@name 3661 \lst@AddToHookExe{PreSet}{\global\let\lst@intname\@empty}
\lst@intname 3662 \lst@AddToHook{PreInit}{% \let\lst@arg\lst@intname \lst@ReplaceIn\lst@arg\lst@filenamerpl
3663 \global\let\lst@name\lst@arg \global\let\lstname\lst@name}

Use of \lst@ReplaceIn removes a bug first reported by Magne Rudshaug. Here is the replacement list.
\def\lst@filenamerpl{\textunderscore $	extdollar -	extendash$}
\l@lstlisting prints one ‘lol’ line.
3666 \def\l@lstlisting#1#2{\@dottedtocline{1}{1.5em}{2.3em}{#1}{#2}}

\lstlistlistingname contains simply the header name.
3667 \lst@UserCommand\lstlistlistingname{Listings}
\lstlistoflistings We make local adjustments and call \tableofcontents. This way, redefinitions of that macro (e.g. without any \MakeUppercase inside) also take effect on the list of listings.
3668 \lst@UserCommand\lstlistoflistings{\bgroup
3669 \let\contentsname\lstlistlistingname
3670 \let\lst@temp\@starttoc \def\@starttoc##1{\lst@temp{lol}}% 
3671 \tableofcontents \egroup}

For KOMA-script classes, we define it a la KOMA thanks to a bug report by Tino Langer. Markus Kohm suggested a much-improved version of this, which also works with the float package. The following few comments are from Markus.

Make use of \float@listhead if defined (e.g. using float or KOMA-Script)
\@ifundefined{float@listhead}{}{%
\renewcommand*{\lstlistoflistings}{% 
\begingroup
Switch to one-column mode if the switch for switching is available.
3675 \@ifundefined{@restonecoltrue}{}{% 
3676 \if@twocolumn \@restonecoltrue\onecolumn 
3677 \else \@restonecolfalse 
3678 \fi 
3679 \}\fi 
3680 \float@listhead{\lstlistlistingname}%
3681 \@ifundefined{@restonecoltrue}{}{% 
3682 \@parfillskip to 0pt (should be!), \parindent to 0pt (better but not always needed), \parfillskip to 0pt plus 1fil (should be!).
3683 \@parfillskip to 0pt plus 1fil% 
3684 \@starttoc{lol}%
3685 \@ifundefined{@restonecoltrue}{}{% 
3686 \if@restonecol\twocolumn\fi 
3687 \} 
3688 \} 
3689 \endgroup

195
The `float` package defines a generic way for packages to add things (such as chapter names) to all of the lists of floats other than the standard figure and table lists. Each package that defines a list of floats adds a command to `\float@addtolists`, and then packages (such as the KOMA-script document classes) which wish to add things to all lists of floats can then use it, without needing to be aware of all of the possible lists that could exist. Thanks to Markus Kohm for the suggestion.

Unfortunately, `float` defines this with `\newcommand`; thus, to avoid conflict, we have to redefine it after `float` is loaded. `\AtBeginDocument` is the easiest way to do this. Again, thanks to Markus for the advice.

```
\AtBeginDocument{\
  \ifundefined{float@addtolists}\
    {\gdef\float@addtolists#1{\addtocontents{lol}{#1}}}\
  {\let\orig@float@addtolists\float@addtolists\
    \gdef\float@addtolists#1{\addtocontents{lol}{#1}\orig@float@addtolists{#1}}}}\
```

18.4 Inline listings

18.4.1 Processing inline listings

`\lstinline` In addition to `\lstset`, we use `boxpos=b` and `flexiblecolumns`. I’ve inserted `\leavevmode` after bug report from Michael Weber. Olivier Lecarme reported a problem which has gone after removing `\let \lst@newlines \@empty` (now \lst@newlines is a counter!). Unfortunately I don’t know the reason for inserting this code some time ago! At the end of the macro we check the delimiter.

```
\newcommand{\lstinline}[1][]{\
  \leavevmode\bgroup % \hbox\bgroup --> \bgroup\def\lst@boxpos{b}\
  \lsthk@PreSet\lstset{flexiblecolumns,#1}\
  \lsthk@TextStyle\@ifnextchar\bgroup{\
    Luc Van Eycken reported, that the experimental implementation of `\lstinline` with braces instead of characters surrounding the source code resulted in an error if used in a tabular environment. He found that this error comes from the master counter (cf. appendix D (Dirty Tricks), item 5. (Brace hacks), of the TeXbook (p. 385-386)). Adding the following line at this point
    \% \ifnum'{=0}\fi% \%

remedies the wrong behaviour. But Qing Lee pointed out, that this breaks code like the one showed in 7.1 on 56 and proposed another solution which in turn broke the code needed by Luc:

\%
\renewcommand{\lstinline}[1][]{\
  \leavevmode\bgroup % \hbox\bgroup --> \bgroup\def\lst@boxpos{b}\
```

196
So finally the old code comes back and the people, who need a \lstinline with
braces, should use the workaround from section 7.1 on page 56.

\lst@InlineM treat the cases of ‘normal’ inlines and inline listings inside an argument. In the
first case the given character ends the inline listing and EOL within such a listing
immediately ends it and produces an error message.

\lst@InlineJ In the other case we get all characters up to \#1, make these characters active,
execute (typeset) them and end the listing (all via temporary macro). That’s all
about it.

\lst@InlineG is experimental.

197
\fi
\expandafter\lst@next\fi\
\def\lst@InlineGJEnd{\lst@arg\lst@DeInit\egroup}
\def\lst@InlineGJRead#1{%\lccode'~'=\lowercase{\lst@lAddTo\lst@arg~}\
st@InlineGJ}
\def\lst@InlineGJReadSp#1{\lccode'~='\lowercase{\lst@lAddTo\lst@arg~}\
st@InlineGJ#1}
\def\lst@shortinlinedef{\lstinline[#1]}\
\lstMakeShortInline@}
\def\lstMakeShortInline@#1{\expandafter\ifx\csname lst@ShortInlineOldCatcode\string#1\endcsname\relax
\lst@shortlstinlineinfo{Made }{#1}\
\lst@add@special{#1}
The character's current catcode is stored in \lst@ShortInlineOldCatcode\langle c\rangle.
\expandafter\xdef\csname lst@ShortInlineOldCatcode\string#1\endcsname{\the\catcode'#1}
The character is spliced into the definition using the same trick as used in \verb\ (for instance), having activated ~ in a group.
\begingroup\catcode'~\active 
\lccode'~'=\lowercase{\lst@shortinlinedef#1}}\
\endgroup
Finally the character is made active.
\global\catcode'~\active
\PackageError{Listings}{\string\lstMakeShorterInline\ definitions cannot be nested}{Use \string\lstDeleteShortInline first.}
This helper macro adds its argument to the \dospecials macro which is conventionally used by verbatim macros to alter the catcodes of the currently active characters. We need to add \do{⟨c⟩} to the expansion of \dospecials after removing the character if it was already there to avoid multiple copies building up should \lstMakeShortInline not be balanced by \lstDeleteShortInline (in case anything that uses \dospecials cares about repetitions).

Similarly we have to add \@makeother⟨c⟩ to \@sanitize (which is used in things like ” to re-catcode all special characters except braces).

The inverse of \lst@add@special is slightly trickier. \do is re-defined to expand to nothing if its argument is the character of interest, otherwise to expand simply to the argument. We can then re-define \dospecials to be the expansion of itself. The space after =’##1 prevents an expansion to \relax!
Fixing `\@sanitize` is the same except that we need to re-define `\@makeother` which obviously needs to be done in a group.

```latex
\begingroup
\def\@makeother##1{\ifnum'#1='##1 \else \noexpand\@makeother\noexpand##1\fi}\
def\@sanitize{\@sanitize}\
\endgroup}
```

18.5 The input command

The macro appends a slash to a path if necessary.

```latex
\def\lst@MakePath#1{\ifx\@empty#1\@empty\else\lst@MakePath@#1/\@nil/\fi}\
def\lst@MakePath@#1/{#1/\lst@MakePath@@}\
def\lst@MakePath@@#1/{\ifx\@nil#1\expandafter\@gobble\else \ifx\@empty#1\else #1/\fi \fi}\
\xdef\@sanitize{\@sanitize}\
\endgroup}
```

Now we can empty the path or use `\lst@MakePath`.

```latex
\lst@Key{inputpath}{}{\edef\lst@inputpath{\lst@MakePath{#1}}}\
```

We use `\lst@doendpe` to remove indentation at the beginning of the next line—except there is an empty line after `\lstinputlisting`. Bug was reported by David John Evans and David Carlisle pointed me to the solution.

```latex
\def\lst@MissingFileError#1#2{\
def\reserved@a{\noexpand\lst@MissingFileError\
{\filename@area\filename@base}\
{\ifx\filename@ext\relax tex\else\filename@ext\fi}}}\
\reserved@a
```

18.5 The input command

The input command

```latex
\lst@inputlisting inputs the listing or asks the user for a new file name.

```latex
\def\lstinputlisting{\begingroup \lst@setcatcodes \lst@inputlisting}\
def\lst@inputlisting[2][2]{\endgroup\lst@set{#1}\
\expandafter\lstset\expandafter{\lst@set}\
\IfFileExists{\lst@inputpath#2}{}{\filename@parse{\lst@inputpath#2}\
\edef\reserved@a{\noexpand\lst@MissingFileError\
{\filename@area\filename@base}\
{\ifx\filename@ext\relax tex\else\filename@ext\fi}}}\
\reserved@a
```

We use `\lst@doendpe` to remove indentation at the beginning of the next line—except there is an empty line after `\lstinputlisting`. Bug was reported by David John Evans and David Carlisle pointed me to the solution.

`\lst@MissingFileError` is a derivation of \LaTeX's `\missingfileerror`. The parenthesis have been added after Heiko Oberdiek reported about a problem discussed on TEX-D-L.

```latex
\def\lst@MissingFileError#1#2{\endgroup
```

200
Typing x or X exits.

In all other cases we try the new file name.

\filenames@parse\@gtempa
\edef\filename@ext{\if\filename@ext\relax#2\else\filename@ext\fi}
\edef\reserved@a{\IfFileExists{\filename@area\filename@base.\filename@ext}{\lst@InputListing{\filename@area\filename@base\filename@ext}}{\lst@MissingFileError{\filename@area\filename@base}{\filename@ext}}}
\expandafter\reserved@a

\lst@ifdraft makes use of \lst@ifprint. Enrico Straube requested the final option.

\let\lst@ifdraft\iffalse
\DeclareOption{draft}{\let\lst@ifdraft\iftrue}
\DeclareOption{final}{\let\lst@ifdraft\iffalse}
\lst@AddToHook{PreSet}{\if\lst@ifdraft
\let\lst@ifprint\iffalse\@gobbletwo\fi\fi}
\lst@InputListing The one and only argument is the file name, but we have the ‘implicit’ argument \lst@set. Note that \lst@Init takes \relax as argument.

\def\lst@InputListing#1{\begingroup\lsthk@PreSet \gdef\lst@intname{#1}\%\expandafter\lstset\expandafter{\lst@set}\%\lsthk@DisplayStyle\catcode\active=\active\lst@Init\relax \let\lst@gobble\z@\lst@SkipToFirst\lst@InputListing\def\lst@next{\input{#1}}\else \let\lst@next\@empty \fi\lst@next\lst@DeInit\endgroup}

The line \catcode\active=\active, which makes the CR-character active, has been added after a bug report by Rene H. Larsen.

\lst@SkipToFirst The end of line character either processes the listing or is responsible for dropping lines up to first printing line.

\lst@ifdraft
We drop the input and redefine the end of line characters.

We just look whether to drop more lines or to leave the mode which restores the definition of chr(13) and chr(10).

18.6 The environment
18.6.1 Low-level processing
\lstenv@DroppedWarning gives a warning if characters have been dropped.

\lstenv@Process We execute \lstenv@ProcessM or \lstenv@ProcessJ according to whether we find an active EOL or a nonactive "\^J. We make no extra \lstenv@ProcessM definition since there is nothing to do at all if we've found an active EOL.

\lstenv@ProcessJ Now comes the horrible scenario: a listing inside an argument. We've already worked in section 13.4 for this. Here we must get all characters up to "end environment". We distinguish the cases 'command fashion' and 'true environment'.
The first case is pretty simple: The code is terminated by \end{name of environment}. Thus we expand that control sequence before defining a temporary macro, which gets the listing and does all the rest. Back to the definition of \lst@ProcessJ we call the temporary macro after expanding \fi.

\begin{Verbatim}
\expandafter\def\expandafter\lst@temp\expandafter##1\%
\csname end\lstenv@name\endcsname
\end{Verbatim}

We must append an active backslash and the ‘end string’ to \lst@arg. So all (in fact most) other processing won’t notice that the code has been inside an argument. But the EOL character is chr(10)=‘\’ now and not chr(13).

The ‘true environment’ case is more complicated. We get all characters up to an \end and the following argument. If that equals \lstenv@name, we have found the end of environment and start typesetting.

\begin{Verbatim}
def\lstenv@ProcessJEnv#1\end#2{\def\lst@temp{#2}\%
\ifx\lstenv@name\lst@temp
\lst@InsideConvert{#1}\%
\expandafter\lstenv@ProcessJ@
\else
\lst@InsideConvert{#1\%\end\{#2\}}\%
\expandafter\lstenv@ProcessJEnv}
\end{Verbatim}

\lstenv@backslash Coming to a backslash we either end the listing or process a backslash and insert the eaten characters again.

\begin{Verbatim}
def\lstenv@backslash\%
\lst@IfNextChars\lstenv@endstring
{\lstenv@End\%
{\expandafter\lst@backslash \lst@eaten}}\%
\expandafter\lstenv@ProcessJEnv
\end{Verbatim}

\lstenv@End This macro has just been used and terminates a listing environment: We call the ‘end environment’ macro using \end or as a command.

\begin{Verbatim}
def\lstenv@End\%
\ifx\@currenvir\lstenv@name
\edef\lst@next{\noexpand\end{\lstenv@name}}\%
\end{Verbatim}

203
18.6.2 Defining new environments

\lstnewenvironment Now comes the main command. We define undefined environments only. On the parameter text \#1\#2 (in particular the last sharp) see the paragraph following example 20.5 on page 204 of ‘The \TeX{}book’.

A lonely ‘end environment’ produces an error.

The ‘main’ environment macro defines the environment name for later use and calls a submacro getting all arguments. We open a group and make EOL active. This ensures \@ifnextchar{ not to read characters of the listing—it reads the active EOL instead.

The submacro is defined via \new@command. We misuse \l@ngrel@x to make the definition \global and refine \LaTeX{}’s \@xargdef.

First we execute \#4=\langle begin code \rangle. Then follows the definition of the terminating string (end\lstlisting or endlstlisting, for example):

We redefine (locally) ‘end environment’ since ending is legal now. Note that the redefinition also works inside a \TeX{} comment line.

\lst@doendpe again removes the indentation problem.

Finally we start the processing. The \lst@EndProcessListing assignment has been moved in front of \lst@Init after a bug report by Andreas Deininger.
This is a derivation of \LaTeX's \@xargdef. We expand the submacro's name, use \gdef instead of \def, and hard code a kind of \@protected@testopt.

\lstenv@xargdef\begin{quote}
\begin{verbatim}
def\lstenv@xargdef#1{
  \expandafter\lstenv@xargdef@\csname\string#1\endcsname#1}
def\lstenv@xargdef@#1#2[#3][#4]{% 
  \@ifdefinable#2{\%\gdef#2{%\ifx\protect\@typeset@protect\expandafter\lstenv@testopt\else\@x@protect#2\fi\#1\{%#4}\}}\@yargdef\#1\{%tw@\{%#3\{%#5\}}}
\end{verbatim}
\end{quote}

The difference between this macro and \@testopt is that we temporarily reset the catcode of the EOL character \texttt{\textasciitilde M} to read the optional argument.

\begin{quote}
\begin{verbatim}
@\long\def\lstenv@testopt#1#2{%\long\charop{\ifnextchar[\relax \lstenv@testopt@#1}\{#1[#2]\}}
\end{verbatim}
\end{quote}

We use the temporary definition

\begin{quote}
\begin{verbatim}
@\long\def\lst@temp##1\{\content of \lstenv@endstring\}\lstenv@End
\end{verbatim}
\end{quote}

which gobbles all characters up to the end of environment and finishes it.
\ltestenv@Error is called by a lonely ‘end environment’.

\lst@TestEOLChar Here we test for the two possible EOL characters.

\lstlisting The awkward work is done, the definition is quite easy now. We test whether
the user has given the name argument, set the keys, and deal with continued line
numbering.

\begin{lstsample}(point list){{(left)}}{{(right)}}\end{lstsample}

\begin{lstxsample}(point list)\end{lstxsample}

The material in between is (a) added to the left side of the next lstsample
environment and (b) typeset verbatim using the whole line width.

19 Documentation support
The \langle name \rangle environment can be used in the same way as ‘macro’. The provided(!) definitions \Print \langle short name \rangle \Name and \SpecialMain \langle short name \rangle \Index control printing in the margin and indexing as the defaults \PrintMacroName and \SpecialMainIndex do.

This command is used to define the ‘aspect’ and ‘\lstkey’ environments.

\macroargs environment

This ‘enumerate’ environment uses as labels ‘\#1 =’, ‘\#2 =’, and so on.

\TODO environment

\ALTERNATIVE environment

\REMOVED environment

\OLDDEF environment

These environments enclose comments on ‘to do’s’, alternatives and removed or old definitions.

\lstscanlanguages \langle list macro \rangle \{ \langle input files \rangle \} \{ \langle don’t input \rangle \}

scans \{ \langle input files \rangle \} \{ \langle don’t input \rangle \} for language definitions. The available languages are stored in \langle list macro \rangle using the form \langle language \rangle (\langle dialect \rangle)

\lstprintlanguages \langle list macro \rangle

prints the languages in two column format.

and a lot of more simple commands.

19.1 Required packages

Most of the ‘required’ packages are optional. Stephan Hennig noted a bug where \ifalgorithmic conflicts with an update to algorithmic.sty, so this has been changed to \ifalgorithmicpkg.

\let \lstdoc@currversion \fileversion
\RequirePackage[writefile]{listings}[2004/09/07]
\newif\iffancyvrb \IfFileExists{fancyvrb.sty}{\fancyvrbtrue}{}
\newif\ifcolor \IfFileExists{color.sty}{\colortrue}{}
\lst@false
\newif\ifhyper
\@ifundefined{pdfoutput}{\ifnum\pdfoutput>\z@ \lst@true \fi}
\@ifundefined{VTeXversion}{\ifnum\OpMode>\z@ \lst@true \fi}
\lst@if \IfFileExists{hyperref.sty}{\hypertrue}{}i
\newif\ifalgorithmicpkg \IfFileExists{algorithmic.sty}{\algorithmicpkgtrue}{}
\newif\iflgrind \IfFileExists{lgrind.sty}{\lgrindtrue}{}
\iffancyvrb \RequirePackage{fancyvrb}\fi
\ifhyper \RequirePackage[ colorlinks ]{hyperref}\else
19.2 Environments for notes

We begin with two simple definitions...

```latex
\begin{quote}
\begin{footnotesize}
\begin{Todo}{To do}
\end{Todo}
\end{footnotesize}
\end{quote}
```

...used to define some environments.

```latex
\begin{ALTERNATIVE}{Alternative}
\end{ALTERNATIVE}
```

```latex
\begin{REMOVED}{Removed}
\end{REMOVED}
```

```latex
\begin{OLDDEF}{Old definition}
\end{OLDDEF}
```

The environment uses `\@listi`.

```latex
\begin{Advise}{par}
\begin{list}{\labeladvise}
\setlength{\labelsep}{0pt}
\setlength{\labelwidth}{0pt}
\setlength{\itemindent}{-\labelwidth}
\setlength{\topsep}{0pt}
\setlength{\parskip}{0pt}
\setlength{\parindent}{0pt}
\setlength{\itemsep}{0pt}
\setlength{\labelsep}{0pt}
\setlength{\listparindent}{0pt}
\setlength{\leftmargin}{0pt}
\setlength{\rightmargin}{0pt}
\setlength{\totalleftmargin}{0pt}
\setlength{\totalwidth}{0pt}
\setlength{\marginparwidth}{0pt}
\setlength{\marginparsep}{0pt}
\setlength{\marginparpush}{0pt}
\setlength{\labelsep}{0pt}
\end{list}
\end{Advise}
```

This environment uses `\list` with a special `\makelabel`, ...

```latex
\begin{Syntax}{par}
\begin{list}{\labeladvise}
\setlength{\labelsep}{0pt}
\setlength{\labelwidth}{0pt}
\setlength{\itemindent}{-\labelwidth}
\setlength{\topsep}{0pt}
\setlength{\parskip}{0pt}
\setlength{\parindent}{0pt}
\setlength{\itemsep}{0pt}
\setlength{\labelsep}{0pt}
\setlength{\listparindent}{0pt}
\setlength{\leftmargin}{0pt}
\setlength{\rightmargin}{0pt}
\setlength{\totalleftmargin}{0pt}
\setlength{\totalwidth}{0pt}
\setlength{\marginparwidth}{0pt}
\setlength{\marginparsep}{0pt}
\setlength{\marginparpush}{0pt}
\setlength{\labelsep}{0pt}
\end{list}
\end{Syntax}
```

...which is defined here. The comma separated items are placed as needed.
\alternative  iterates down the list and inserts vertical rule(s).
\def\alternative#1{\lst@true \alternative@#1,\relax}
\def\alternative@#1,{{\ifx\relax#1\@empty\expandafter\@gobble\else\fi\ifx\@empty#1\@empty\else\lst@if \lst@false \else $\vert$\fi\textup{\texttt{#1}}\fi\alternative@}

19.3 Extensions to doc
\m@cro@  We need a slight modification of doc’s internal macro. The former argument #2 has become #3. This change is not marked below. The second argument is now \langle short name \rangle.
\long\def\m@cro@#1#2#3{\endgroup \topsep\MacroTopsep \trivlist
    \edef\saved@macroname{\string#3}\
    \def\makelabel##1{\llap{##1}}\
    \if@inlabel
        \let\@tempa\@empty \count@\macro@cnt
        \loop \ifnum\count@>\z@ \edef\@tempa{\@tempa\hbox{\strut}}\advance\count@\m@ne \repeat
        \edef\makelabel##1{\llap{\vtop to\baselineskip{\@tempa\hbox{##1}\vss}}}\
        \advance \macro@cnt \@ne
    \else \macro@cnt\@ne \fi
    \edef\@tempa{\noexpand\item[\%\expandafter
        \csname Print#2Name\endcsname \string#3]
        \SpecialMainIndex{#3} \nobreak
        \DoNotIndex{#3} \else
        \noexpand\PrintMacroName
        \\else
        \expandafter\noexpand\csname Print#2Name\endcsname \string#3 \global\advance\c@CodelineNo\@ne
        \hfill \SpecialMainIndex{#3} \nobreak
        \DoNotIndex{#3} \else
        \expandafter\noexpand\csname Print#2Name\endcsname \string#3 \fi
    \fi

The next line has been modified.
Ditto.
\csname SpecialMain#2Index\endcsname{#3}\nobreak % MODIFIED
\fi
\global\advance\c@CodelineNo\m@ne
\ignorespaces}
\macro These two definitions need small adjustments due to the modified \m@cro@.
\environment The environment and its ‘print’ and ‘index’ commands.
\newdocenvironment This command simply makes definitions similar to ‘environment’ and provides the printing and indexing commands.
\newcounter{argcount}
\def\labelargcount{\texttt{\#\arabic{argcount}}\hskip\labelsep$=$}
\def\macroargs{\list\labelargcount
{\usecounter{argcount}\leftmargin=2\leftmargin
\parsep \z@ \parskip\z@ \itemsep\z@ \parskip\z@ \itemsep\z@}
\def\makelabel##1{\hss\llap{##1}}}
\def\endmacroargs{\endlist\@endparenv}
\labelargcount We just allocate a counter and use \TeX’s \list to implement this environment.
19.4 The \texttt{lstsample} environment

\texttt{lstsample} We store the verbatim part and write the source code also to file.

\begin{verbatim}
\lst@RequireAspects{writefile}
\newbox\lst@samplebox
\lstnewenvironment{lstsample}[3]{\@empty}{\@empty}{\gdef\lst@sample{#2}\
\setkeys{lst}{language={},style={},tabsize=4,gobble=5,\
  basicstyle=\small	tfamily,basewidth=0.51em,point={#1}
  #3}\
\lst@BeginAlsoWriteFile{\jobname.tmp}}\lst@EndWriteFile\
\end{verbatim}

Now \texttt{\lst@samplebox} contains the verbatim part. If it’s too wide, we use atop
and below instead of left and right.

\begin{verbatim}
\ifdim \wd\lst@samplebox>.5\linewidth\begin{center}\
\hbox to\linewidth{\box\lst@samplebox\hss}\end{center}\
\else\begin{center}\
\begin{minipage}{0.45\linewidth}\lst@sampleInput\end{minipage}\
\qquad\begin{minipage}{0.45\linewidth}\
\hbox to\linewidth{\box\lst@samplebox\hss}\end{minipage}\
\end{center}\fi
\end{verbatim}

The new keyword class \texttt{point}.

\begin{verbatim}
\lst@InstallKeywords{p}{point}{pointstyle}\relax{keywordstyle}{}ld
\end{verbatim}

\texttt{lstxsample} Omitting \texttt{\lst@EndWriteFile} leaves the file open.

\begin{verbatim}
\lstnewenvironment{lstxsample}[1]{\begingroup\setkeys{lst}{belowskip=-\medskipamount,language={},style={},\
  tabsize=4,gobble=5,basicstyle=\small	tfamily,\
  basewidth=0.51em,point={#1}}\lst@BeginAlsoWriteFile{\jobname.tmp}}\endgroup\
\end{verbatim}

\texttt{\lst@sampleInput} inputs the ‘left-hand’ side.

\begin{verbatim}
\def\lst@SampleInput{\MakePercentComment\catcode`\^^M=10\relax\small\lst@sample{\setkeys{lst}{SelectCharTable=\lst@ReplaceInput{\^^I}\
\lst@ProcessTabulator}}\leavevmode \input{\jobname.tmp}}\MakePercentIgnore
\end{verbatim}
19.5 Miscellaneous

Sectioning and cross referencing  We begin with a redefinition paragraph.

We introduce \lstref which prints section number together with its name.

Moreover we adjust the table of contents. The \phantomsection before adding the contents line provides hyperref with an appropriate destination for the contents line link, thereby ensuring that the contents line is at the right level in the PDF bookmark tree.

Indexing  The ‘user’ commands. \rstyle is defined below.
One of the two yet unknown ‘index’-macros is empty, the other looks up the aspect name for the given argument.

The key/command to aspect relation is defined near the top of this file using the following command. In future the package should read this information from the aspect files.

This relation is also good to print all keys and commands of a particular aspect ...

... or to check the reference. Note that we’ve defined \lstkandc@{name} in \lstaspectindex.

Unique styles
The color mainly for keys and commands in the reference guide.

\ifcolor
  \definecolor{darkgreen}{rgb}{0,0.5,0}
  \def\rstyle{\color{darkgreen}}
\else
  \let\rstyle\empty
\fi

Commands for credits and helpers

There are two commands for credits and helpers:

1. \lstthanks is used to put a name of a contributor into the section “Closing and credit”. It has two arguments: \#1 is the name, \#2 the email address—the email address is not shown.

2. \lsthelper must be used in the text to show the name of the helper (argument \#1), the date of the contribution (argument \#2) and a short text about the contribution (argument \#3). Only the first argument is printed.

Languages and styles

\lstdefinestyle{doc}{%
  morekeywords={alfa,and,array,begin,boolean,byte,char,const,div,\%
  do,downto,else,end,false,file,for,function,get,goto,if,in,\%
  integer,label,main,mod,new,not,or,pack,packed,page,program,\%
  procedure,put,read,readln,real,record,repeat,reset,rewrite,set,\%
  text,then,to,true,type,unpack,until,var,while,with,write,writeln},\%
  sensitive=false,\%
  morecomment=[s]{\{\(\star\}\{\(\star\)}},\%
  morecomment=[s]{\{/\}\}},\%
  morestring=[d]{'\}\}'}

\lstdefinestyle{doc}{%
  morekeywords={alfa,and,array,begin,boolean,byte,char,const,div,\%
  do,downto,else,end,false,file,for,function,get,goto,if,in,\%
  integer,label,main,mod,new,not,or,pack,packed,page,program,\%
  procedure,put,read,readln,real,record,repeat,reset,rewrite,set,\%
  text,then,to,true,type,unpack,until,var,while,with,write,writeln},\%
  sensitive=false,\%
  morecomment=[s]{\{\(\star\}\{\(\star\)}},\%
  morecomment=[s]{\{/\}\}},\%
  morestring=[d]{'\}\}'}

\lstdefinestyle{doc}{%
  morekeywords={alfa,and,array,begin,boolean,byte,char,const,div,\%
  do,downto,else,end,false,file,for,function,get,goto,if,in,\%
  integer,label,main,mod,new,not,or,pack,packed,page,program,\%
  procedure,put,read,readln,real,record,repeat,reset,rewrite,set,\%
  text,then,to,true,type,unpack,until,var,while,with,write,writeln},\%
  sensitive=false,\%
  morecomment=[s]{\{\(\star\}\{\(\star\)}},\%
  morecomment=[s]{\{/\}\}},\%
  morestring=[d]{'\}\}'}

\lstdefinestyle{}%
19.6 Scanning languages

\lstscanlanguages  We modify some internal definitions and input the files.

\do creates a box of width 0.5\linewidth or \linewidth depending on how wide
the argument is. This leads to ‘two column’ output. The other main thing is
sorting the list and begin with the output.

\lstprintlanguages  We get and define the current language and . . .

... gather all available dialect of this language (note that the list has been sorted)
Here we take care of default dialects.
\def\lst@PrintDialect#1{%
\lst@NormedDef\lst@temp{#1}%
\expandafter\ifx\csname\@lst dd@\lst@language\endcsname\lst@temp
\texttt{\underline{#1}}%
\else
\texttt{#1}%
\fi
\expandafter\lst@PrintDialects\relax%
\fi}

19.7 Bubble sort
\lst@IfLE ⟨string 1⟩\relax⟨empty⟩⟨string 2⟩\relax⟨empty⟩⟨then⟩⟨else⟩. If ⟨string 1⟩ ≤ ⟨string 2⟩, we execute ⟨then⟩ and ⟨else⟩ otherwise. Note that this comparison is case insensitive.
\lst@BubbleSort is in fact a derivation of bubble sort.
We ‘bubble sort’ the first, second, ... elements and ...
... then the second, third, ... elements until no elements have been swapped.
But the bubbles rise only one step per call. Putting the elements at their top most place would be inefficient (since \TeX had to read much more parameters in this case).
20 Interfaces to other programs

20.1 0.21 compatibility

Some keys have just been renamed.

\lst@BeginAspect{0.21}
\lst@Key{labelstyle}{}{\def\lst@numberstyle{#1}}
\lst@Key{labelsep}{10pt}{\def\lst@numbersep{#1}}
\lst@Key{labelstep}{0}{%\ifnum #1=\z@ \KV@lst@numbers{none}\
\else \KV@lst@numbers{left}\fi}
\lst@Key{firstlabel}{}{\def\lst@firstnumber{#1}}
\lst@Key{advancelabel}{}{\def\lst@advancenumber{#1}}
\let\c@lstlabel\c@lstnumber
\lst@AddToHook{Init}{\def\thelstnumber{\thelstlabel}}
\newcommand*{\thelstlabel}{\@arabic{\c@lstlabel}}
\lst@Key{first}{\def\lst@firstline{#1}}
\lst@Key{last}{}{\def\lst@lastline{#1}}
\lst@Key{framerulewidth}{.4pt}{\def\lst@framerulewidth{#1}}
\lst@Key{framerulesep}{2pt}{\def\lst@rulesep{#1}}
\lst@Key{frametextsep}{3pt}{\def\lst@frametextsep{#1}}
\lst@Key{framerulecolor}{}{\lstKV@OptArg\[\]{#1}\
{\ifx\@empty##2\@empty\let\lst@rulecolor\@empty\else\
\ifx\@empty##1\@empty\def\lst@rulecolor{\color{##2}}\else\
\def\lst@rulecolor{\color[#1]{##2}}\fi\fi}}
\lst@Key{backgroundcolor}{}{\lstKV@OptArg\[\]{#1}\
{\ifx\@empty##2\@empty\let\lst@bkgcolor\@empty\else\
\ifx\@empty##1\@empty\def\lst@bkgcolor{\color{##2}}\else\
\def\lst@bkgcolor{\color[#1]{##2}}\fi\fi}}
\lst@Key{framespread}{\z@}{\def\lst@framespread{#1}}
\lst@AddToHook{PreInit}{\@tempdima\lst@framespread\relax\divide\@tempdima\tw@\edef\lst@framextopmargin{\the\@tempdima}}
\lst@Key{backgroundcolor}{}{\lstKV@OptArg\[\]{#1}\
{\ifx\@empty##2\@empty\let\lst@bkgcolor\@empty\else\
\ifx\@empty##1\@empty\def\lst@bkgcolor{\color{##2}}\else\
\def\lst@bkgcolor{\color[#1]{##2}}\fi\fi}}
\lst@Key{framespread}{\z@}{\def\lst@framespread{#1}}
\lst@BeginToHook{PreInit}{\@tempdima\lst@framespread\relax\divide\@tempdima\tw@\edef\lst@framextopmargin{\the\@tempdima}}
\let\lst@framexrightmargin=\lst@framextopmargin
\let\lst@framexbottommargin=\lst@framextopmargin
\advance\@tempdima\lst@xleftmargin\relax\edef\lst@framexleftmargin{\the\@tempdima}
Harald Harders had the idea of two spreads (inner and outer). We either divide the dimension by two or assign the two dimensions to inner- and outerspread.

```latex
\newdimen\lst@innerspread \newdimen\lst@outerspread
\lst@Key{spread}{\z@,\z@}{\lstKV@CSTwoArg{#1}%
   \ifx\@empty##1\@empty
      \divide\lst@innerspread\tw@\relax
      \lst@outerspread\lst@innerspread
   \else
      \lst@outerspread##1\relax
   \fi}
\lst@AddToHook{BoxUnsafe}{\lst@outerspread\z@ \lst@innerspread\z@}
\lst@Key{wholeline}{false}[t]{\lstKV@SetIf{#1}\lst@ifresetmargins}
\lst@Key{indent}{\z@}{\def\lst@xleftmargin{#1}}
\lst@AddToHook{PreInit}{\lst@innerspread=-\lst@innerspread
   \lst@outerspread=-\lst@outerspread
   \ifodd\c@page \advance\lst@innerspread\lst@xleftmargin
   \else \advance\lst@outerspread\lst@xleftmargin \fi
   \ifodd\c@page
      \edef\lst@xleftmargin{\the\lst@innerspread}\
      \edef\lst@xrightmargin{\the\lst@outerspread}\
   \else
      \edef\lst@xleftmargin{\the\lst@outerspread}\
      \edef\lst@xrightmargin{\the\lst@innerspread}\
   \fi}
\lst@Key{defaultclass}{\relax}{\def\lst@classoffset{#1}}
\lst@Key{stringtest}{\relax}{\if\lst@iffancyvrb\fi}
\lst@BeginAspect{fancyvrb}
\lst@EndAspect
\langle 0.21 \rangle
```

20.2 fancyvrb

Denis Girou asked whether fancyvrb and listings could work together.

```latex
\lst@Key{fancyvrb}{\relax}{\if\lst@iffancyvrb\fi}
\if\lst@iffancyvrb\lst@RequireAspects{fancyvrb}\fi
\langle kernel \rangle
\lst@BeginAspect{fancyvrb}
\lst@EndAspect
```

We end the job if fancyvrb is not present.
We assign the correct \FancyVerbFormatLine macro.
\gdef\lstFV@fancyvrb{% 
\lst@iffancyvrb \ifx\FancyVerbFormatLine\lstFV@FancyVerbFormatLine\else \let\lstFV@FVFL\FancyVerbFormatLine \let\FancyVerbFormatLine\lstFV@FancyVerbFormatLine \fi \else \ifx\lstFV@FVFL\@undefined\else \let\FancyVerbFormatLine\lstFV@FVFL \let\lstFV@FVFL\@undefined \fi \fi 
}

We initialize things if necessary.
\gdef\lstFV@VerbatimBegin{% 
\ifx\FancyVerbFormatLine\lstFV@FancyVerbFormatLine \lsthk@TextStyle \lsthk@BoxUnsafe \lsthk@PreSet \lst@activecharsfalse \let\normalbaselines\relax \lst@ifresetmargins \advance\linewidth-\@totalleftmargin \fi \lstFV@RestoreData \everypar{} \global\lst@newlines\z@ \lst@mode\lst@nomode \let\lst@entermodes\@empty \lst@InterruptModes 
\par
Rolf Niepraschk reported a bug concerning ligatures to Denis Girou.
}\% D.G. modification begin - Nov. 25, 1998 
\let\@noligs\relax 
\% D.G. modification end 
}

A box and macro must exist after \lst@DeInit. We store them globally.
\gdef\lstFV@VerbatimEnd{% 
\ifx\FancyVerbFormatLine\lstFV@FancyVerbFormatLine \global\setbox\lstFV@gtempboxa\box\@tempboxa \global\let\@gtempa\FV@ProcessLine \lst@mode\lst@Pmode \fi 
}
The \par has been added after a bug report by Peter Bartke.

We insert \lst@WhiteSpaceBegin \lst@WhiteSpaceEnd where necessary.

\lst@Key{fvcmdparams} These keys adjust \lst@Key{morefvcmdparams}, which will be used by the following conversion macro. The base set of commands and parameter numbers was provided by Denis Girou.

\lst@Key{morefvcmdparams}\relax{\lst@lAddTo\lst@FVcmdparams{,#1}}

\lst@FancyVerbFormatLine ‘@’ terminates the argument of \lst@FVConvert. Moreover \lst@ReenterModes and \lst@InterruptModes encloses some code. This ensures that we have same group level at the beginning and at the end of the macro—even if the user begins but doesn’t end a comment, which means one open group. Furthermore we use \vtop and reset \lst@newlines to allow line breaking.

\lst@FVcmdparams\morefvcmdparams We do conversion or...
Since \@ifnextchar\bgroup might fail, we have to use \ifcat here. Bug reported by Denis Girou. However we don’t gobble space tokens as \@ifnextchar does.

\gdef\lst@FVConvert@@{% 
\ifcat\noexpand\@let@token\bgroup \expandafter\lst@FVConvertArg 
\else \expandafter\lst@FVConvert@ \fi}

Coming to such a catcode = 1 character we convert the argument and add it together with group delimiters to \lst@arg. We also add \lst@PrintToken, which prints all collected characters before we forget them. Finally we continue the conversion.

\gdef\lst@FVConvertArg#1{% 
{\let\lst@arg\@empty \lst@FVConvert#1\@nil} 
\global\let\@gtempa\lst@arg} 
\lst@lExtend\lst@arg{"\@gtempa\lst@PrintToken}} 
\lst@FVConvert}

\gdef\lst@FVConvert@#1{% 
\ifx\@nil#1\else 
\if\relax\noexpand#1% 
\lst@lAddTo\lst@arg{\lst@OutputLostSpace\lst@PrintToken#1}% 
\else \lccode'~'#1\lowercase{\lst@lAddTo\lst@arg~} \fi 
\expandafter\lst@FVConvert \fi}

Having no \bgroup, we look whether we’ve found the end of the input, and convert one token ((non)active character or control sequence).

\gdef\lst@FVConvert@#1{% 
\ifx\@nil#1\else 
\if\relax\noexpand#1% 
\lst@lAddTo\lst@arg{\lst@OutputLostSpace\lst@PrintToken#1}% 
\else \lccode'~'#1\lowercase{\lst@lAddTo\lst@arg} \fi 
\fi 
\expandafter\lst@FVConvert \fi}

Here we check for registered commands with arguments and set the value of \@tempcnta as required.

\def\lst@temp##1,#1##2,##3##4\relax{% 
\ifx##3\@empty \else \@tempcnta##2\relax \fi}
\expandafter\lst@temp\lst@FVcmdparams,#1\z@,\@empty\relax 
\else \lccode'~'#1\lowercase{\lst@lAddTo\lst@arg} \fi 
\expandafter\lst@FVConvertO@ \fi}

\lst@EndAspect

\section{Omega support}

Ω support looks easy—I hope it works at least in some cases.
\lst@LGGetNames is used to extract the language names from \lst@arg (the LGrind definition).
\lst@LGGetValue returns in \lst@LGValue the value of capability #1 given by the list \lst@arg.
If #1 is not found, we have \lst@if=\iffalse. Otherwise it is true and the “cap=value” pair is removed from the list. First we test for #1 and
\lst@LGGetString does the same for string delimiters, but it doesn’t ‘return’ any value.
\lst@LGComment stores the comment delimiters (enclosed in braces) in #2 if comment of type #1 is
present and not a comment line. Otherwise #2 is empty.
we must check for \e, i.e.
whether we have to use doubled or backslashed stringizer.

get the keywords and values of friends.

Now we get the comment delimiters and use them as single or double comments
according to whether there are two or four delimiters. Note that \lst@LGGetComment
takes care of comment lines.

\lst@LGDefLang defines the language given by \lst@arg, the definition part, and \lst@language@,
the language name. First we remove unwanted stuff from \lst@arg, e.g. we replace
: \ by ::.

Get the keywords and values of friends.

Now we get the comment delimiters and use them as single or double comments
according to whether there are two or four delimiters. Note that \lst@LGGetComment
takes care of comment lines.
Now we parse the stringizers.

\lst@LGGetString s\lst@LGa
\lst@LGGetString l\lst@LGa

We test for the continuation capability and define the language.

\expandafter\xdef\csname\@lst LGlang@\lst@language@\endcsname
{\noexpand\lstset{\lstlang@}}

Finally we inform the user of all ignored capabilities.

\lst@ReplaceInArg\lst@arg{{: :}:}\let\lst@LGvalue\@empty
\expandafter\lst@LGDroppedCaps\lst@arg\relax\relax
\ifx\lst@LGvalue\@empty\else
\PackageWarningNoLine{Listings}{Ignored capabilities for \space \l@language@' are\MessageBreak\lst@LGvalue}\i

\lst@LGDroppedCaps just drops a previous value and appends the next capability name to \lst@LGvalue.
\gdef\lst@LGDroppedCaps#1:#2#3{\ifx#2\relax\lst@RemoveCommas\lst@LGvalue\else\edef\lst@LGvalue{\lst@LGvalue,#2#3}\expandafter\lst@LGDroppedCaps\fi}

\lst@LGReplace We replace ‘escaped :’$\backslash$’ by catcode 11 versions, and other strings by some kind of short versions (which is necessary to get the above definitions work).

\begingroup
\catcode’/=0
\lccode’/\z=:\lccode’/\y=\^\lccode’/\x=$\lccode’/\v=\|\catcode’\z=12\relax
\gdef\lst@LGReplace{/\lst@ReplaceInArg/\lst@arg
{{\:\:\\:}{{}\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}{\::}}}}
\gdef\lst@LGGe{\e}
\endgroup

\lst@LGRead reads one language definition and defines the language if the correct one is found.
We only have to request the language and
\lst@Key{lgrindef}\relax{\{}% \lst@NormedDef\lst@language@{#1}% \begingroup \@ifundefined{lstLGlang@\lst@language@}{\everypar{\lst@LGRead}\catcode'\=12\catcode'\{=12\catcode'\}=12\catcode'%=12\catcode'#=14\catcode'$=12\catcode'^=12\catcode'_=12\relax\input{\lstlgrindeffile}{}}\endgroup
\lsthk@SetLanguage\csname\@lstLGlang@\lst@language@\endcsname}\\
lstlgrindeffile contains just the file name.
\lst@UserCommand{lstlgrindeffile}{lgrindef.}{}\lst@EndAspect/../misc\lst@BeginAspect[keywords]{hyper}determine the macro to set an anchor and a link, respectively.
\lst@Key{hyperanchor}\hyper@@anchor{\let\lst@hyperanchor#1}\lst@Key{hyperlink}\hyperlink{\let\lst@hyperlink#1}Again, the main thing is a special working procedure. First we extract the contents of \lst@token and get a free macro name for this current character string (using prefix \lst@HR@ and a number as suffix). Then we make this free macro equivalent to \@empty, so it is not used the next time.
\lst@UM\@empty xdef\@gtempa{\the\lst@token}\
\endgroup
\lst@GetFreeMacro{lstHR\@gtempa}\
\global\expandafter\let\lst@freemacro\@empty
\@tempcnta is the suffix of the free macro. We use it here to refer to the
last occurrence of the same string. To do this, we redefine the output macro
\lst@alloverstyle to set an anchor ...\n\@tempcntb\@tempcnta \advance\@tempcntb\m@ne
\edef\lst@alloverstyle##1{\let\noexpand\lst@alloverstyle\noexpand\@empty\noexpand\smash{\raise\baselineskip\hbox{\noexpand\lst@hyperanchor{lst.\@gtempa\the\@tempcnta}{}{\relax}}}\ifnum\@tempcnta=\z@ ##1\else\noexpand\lst@hyperlink{lst.\@gtempa\the\@tempcntb}{##1}\fi}\}
od\lst@EndAspect
\langle misc \rangle

21 Epilogue

Each option adds the aspect name to \lst@loadaspects or removes it from that
data macro.
\DeclareOption*{\expandafter\lst@ProcessOption\CurrentOption\relax}
\def\lst@ProcessOption#1#2\relax{\ifx #1!%\lst@DeleteKeysIn\lst@loadaspects{#2}%\else\lst@lAddTo\lst@loadaspects{,#1#2}%\fi}
The following aspects are loaded by default.
\@ifundefined{lst@loadaspects}{\def\lst@loadaspects{strings,comments,escape,style,language,\
keywords,labels,lineshape,frames,emph,index})%\}{
We load the patch file, ...\n\InputIfFileExists{lstpatch.sty}{\{}\}
... process the options, ...\n\let\lst@ifsavemem\iffalse\DeclareOption{savemem}{\let\lst@ifsavemem\iftrue}\DeclareOption{noaspects}{\let\lst@loadaspects\@empty}\ProcessOptions
... and load the aspects.
\lst@RequireAspects\lst@loadaspects\let\lst@loadaspects\@empty
If present we select the empty style and language.

Finally we load the configuration files.

22 History

Only major changes are listed here. Introductory version numbers of commands and keys are in the sources of the guides, which makes this history fairly short.

0.1 from 1996/03/09
- test version to look whether package is possible or not

0.11 from 1996/08/19
- improved alignment

0.12 from 1997/01/16
- nearly ‘perfect’ alignment

0.13 from 1997/02/11
- load on demand: language specific macros moved to driver files
- comments are declared now and not implemented for each language again (this makes the \TeX sources easier to read)

0.14 from 1997/02/18
- User’s guide rewritten, Implementation guide uses macro environment
- (non) case sensitivity implemented and multiple string types, i.e. Modula-2 handles both string types: quotes and double quotes

0.15 from 1997/04/18
- package renamed from listing to listings since the first already exists

0.16 from 1997/06/01
- listing environment rewritten

0.17 from 1997/09/29
- speed up things (quick ‘if parameter empty’, all \texttt{long} except one removed, faster \texttt{\lst@GotoNextTabStop}, etc.)
- improved alignment of wide other characters (e.g. ==)

pre-0.18 from 1998/03/24 (unpublished)
- experimental implementation of character classes

0.19 from 1998/11/09
- character classes and new \texttt{lst}-aspects seem to be good
- user interface uses \texttt{keyval} package
- \texttt{fancyvrb} support

0.20 from 1999/07/12
- new keyword detection mechanism
- new aspects: \texttt{writefile}, \texttt{breaklines}, captions, html
– all aspects reside in a single file and the language drivers in currently two files

0.21 2000/08/23
– completely new User’s guide
– experimental format definitions
– keyword classes replaced by families
– dynamic modes

1.0β 2001/09/21
– key names synchronized with fancyverb
– frames aspect extended
– new output concept (delaying and merging)

1.0 2002/04/01
– update of all documentation sections including Developer’s guide
– delimiters unified

1.1 2003/06/21
– bugfix-release with some new keys

1.2 2004/02/13
– bugfix-release with two new keys and new section 5.7

1.3 2004/09/07
– another bugfix-release with LPPL-1.3-compliance

1.4 2007/02/26
– many bugfixes, and new maintainership
– several new and updated language definitions
– many small documentation improvements
– new keys, multicharacter string delimiters, short inline listings, and more.

1.5 2013/06/27
– new maintainership

1.6 2015/05/05
– add discussion about using \lstinline{(key=value list)}{(source code)}.
– add section “Bugs and workarounds”.

1.7 2018/09/02
– some new or updated language definitions.
– several error corrections.

1.8 from 2019/02/27 on
– corrected and activated the option inputpath.
– some new or updated language definitions.
– several error corrections.
– introduced \lstlistingnamestyle.

References


Index

Symbols

" ........................................ 197
root ........................................ 19
square ........................................ 19
directives .......................... 43
moredirectives ..................... 43

E
emph
deleteemph .......................... 30
emphstyle .......................... 18, 31
emph .................................. 18, 30
moreemph .......................... 30
escape
e scapebegin .......................... 39
escapechar .......................... 39, 57
escapeend .......................... 39
escapeinside .......................... 39, 57
mathescape .......................... 39, 56
texcl .................................. 39, 57
experimental
includerangemarker ................. 52
rangebeginprefix .................. 52
rangebeginsuffix .................. 52
rangedefprefix .................. 52
rangeendsuffix .................. 52
rangeprefix .................. 52
rangesuffix .................. 52

c
comment styles
b .......................... 22
d .......................... 22
is .......................... 23
l .......................... 22
n .......................... 22
s .......................... 22

experimental
includerangemarker ................. 52
rangebeginprefix .................. 52
rangebeginsuffix .................. 52
rangedefprefix .................. 52
rangeendsuffix .................. 52
rangeprefix .................. 52
rangesuffix .................. 52

formats
\lstdefineformat ................. 51
format .......................... 51

frames
bgcolor .......................... 17, 36
fillcolor .......................... 36
frameround .......................... 16, 36
framerule .......................... 36
framesep .......................... 36
frameshape .......................... 37
framerbottommargin ............ 36
framerleftmargin ............ 36
framerightmargin ............ 36
framerbottommargin ............ 36
frame .......................... 16, 35, 36
rulecolor .......................... 36
rulesepcolor .......................... 36
rulesep .......................... 36

D
directives
deletedirectives .................. 44
directivestyle ..................... 30
H

html
markfirstintag .......... 29
tagstyle ................ 29
tag ..................... 44
usekeywordsintag .......... 29

hyper
deletehyperref .......... 49
hyperanchor .............. 49
hyperlink ................ 49
hyperref ................ 49
morehyperref ............. 49

I

index
\lstindexmacro .......... 37
deleteindex .............. 37
indexstyle ................ 19, 37
index ................ 19, 37
moreindex ................ 37

K

kernel
\lstDeleteShortInline ...... 42
\lstMakeShortInline ...... 42
\lstaspectfiles .......... 47
\lstinline ............ 11, 26
\lstinputlisting .......... 4, 27
\lstlistingnamestyle ... 34, 47
\lstlistingname .......... 34, 47
\lstlistoflistings ...... 17, 33
\lstname ................ 34
\lstnewenvironment ...... 41
\letset ................. 10, 26, 27
\thelstlisting .......... 34
abovecaptionskip ...... 34
abovecomment ............ 133
abovecomment ............ 133
aboveescape ............. 157
abovecaption ............ 133
abovecaption ............ 133
alsodigit ................. 122
alsolanguage .......... 158
alsoletter ............... 122
alsoother ................. 122
backgroundcolor .......... 171
breakatwhitespace ...... 169
breakautoindent .......... 169
breakautoindent .......... 169
breaklines .............. 169
caption ................. 184
caption ................. 184
columns ................ 20, 38
deletedelim .............. 31
delim ................... 31
columns ................ 107
comment ................ 133
columns ................ 107
defaultdialect .......... 159
defaultdialect .......... 159
keywords

classoffset ........... 29, 30
deletekeywords ......... 43
deletendkeywords ........ 43
keywordsprefix ........... 43
keywordsstyle ........... 5, 30
keywords ................ 43
morekeywords .......... 21, 43
moreendkeywords ........... 43
ndkeywordstyle ........... 30
ndkeywords ............... 43
otherkeywords ............ 44
keywords ............... 44

defaultdialect ........ 29, 47
delteprocnamekeys ...... 48
indexprocnames ........ 49
moreprocnameskeys ...... 48
procnameskeys ............ 48
procnamekeys ............. 48
procnaminestyle .......... 49

labels

\lstnumber .............. 33
firstnumber ........... 15, 32
numberblanklines ........ 32
numberfirstline ......... 32
numberep .............. 6, 14, 32
numberstyle ............ 6, 14, 32
numbers ............... 6, 14, 32
stepnumber ........... 6, 14, 32

language

\lst@definelanguage ....... 42
\lstalias .......... 42, 43, 47
\lstdefine ...... 42
\lstlanguagefiles ...... 47
\lstloadlanguages ........ 10
alsoalanguage .......... 11, 29
defaultdialect ....... 29, 47
language ............... 11, 28

lgrind

\lstgrindeffile .......... 50
lgrinddef ............. 50

lineshape

breakatwhitespace ...... 35
breakautoindent ...... 35

language

\lst@definelanguage ....... 42
\lstalias .......... 42, 43, 47
\lstdefine ...... 42
\lstlanguagefiles ...... 47
\lstloadlanguages ........ 10
alsoalanguage .......... 11, 29
defaultdialect ....... 29, 47
language ............... 11, 28

lgrind

\lstgrindeffile .......... 50
lgrinddef ............. 50

strings

delestring ............ 23, 45
morestring ............ 21, 45
showstringspaces ...... 5, 32
stringstyle ............ 5, 30
string ................ 45

lgrind

\lstgrindeffile .......... 50
lgrinddef ............. 50

strings

delestring ............ 23, 45
morestring ............ 21, 45
showstringspaces ...... 5, 32
stringstyle ............ 5, 30
string ................ 45

语言

\lst@definelanguage ....... 42
\lstalias .......... 42, 43, 47
\lstdefine ...... 42
\lstlanguagefiles ...... 47
\lstloadlanguages ........ 10
alsoalanguage .......... 11, 29
defaultdialect ....... 29, 47
language ............... 11, 28

lgrind

\lstgrindeffile .......... 50
lgrinddef ............. 50

语言

\lst@definelanguage ....... 42
\lstalias .......... 42, 43, 47
\lstdefine ...... 42
\lstlanguagefiles ...... 47
\lstloadlanguages ........ 10
alsoalanguage .......... 11, 29
defaultdialect ....... 29, 47
language ............... 11, 28

lgrind

\lstgrindeffile .......... 50
lgrinddef ............. 50

休眠

breakatwhitespace ...... 35
breakautoindent ...... 35