

The Listings Package

Copyright 1996–2004

Carsten Heinz <cheinz@gmx.de>

2004/02/13 Version 1.2

Abstract

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

User's guide	4	4.8 Line numbers	31
1 Getting started	4	4.9 Captions	32
1.1 A minimal file	4	4.10 Margins and line shape	33
1.2 Typesetting listings	4	4.11 Frames	34
1.3 Figure out the appearance	6	4.12 Indexing	36
1.4 Seduce to use	7	4.13 Column alignment	36
1.5 Alternatives	8	4.14 Escaping to L ^A T _E X	37
2 The next steps	10	4.15 Interface to fancyvrb	39
2.1 Software license	10	4.16 Environments	40
2.2 Package loading	11	4.17 Language definitions	40
2.3 The key=value interface	12	4.18 Installation	44
2.4 Programming languages	12	5 Experimental features	45
2.5 Special characters	14	5.1 Listings inside arguments	45
2.6 Line numbers	15	5.2 † Export of identifiers	46
2.7 Layout elements	16	5.3 † Hyper references	47
2.8 Emphasize identifiers	18	5.4 Literate programming	47
2.9 Indexing	19	5.5 LGrind definitions	48
2.10 Fixed and flexible columns	20	5.6 † Automatic formatting	48
3 Advanced techniques	21	5.7 Arbitrary linerange markers	50
3.1 Style definitions	21	6 Forthcoming ?	51
3.2 Language definitions	21		
3.3 Delimiters	22	Tips and tricks	51
3.4 Closing and credits	24	7 Troubleshooting	51
		8 How tos	52
Reference guide	25		
4 Main reference	25	Developer's guide	54
4.1 How to read the reference	25	9 Basic concepts	55
4.2 Typesetting listings	26	9.1 Package loading	55
4.3 Space and placement	26	9.2 How to define <code>lst</code> -aspects	57
4.4 The printed range	27	9.3 Internal modes	60
4.5 Languages and styles	28	9.4 Hooks	63
4.6 Figure out the appearance	29	9.5 Character tables	65
4.7 Getting all characters right	30		

9.6 On the output	67	16 Keywords	133
10 Package extensions	68	16.1 Making tests	133
10.1 Keywords and working identifiers	68	16.2 Installing tests	136
10.2 Delimiters	69	16.3 Classes and families	139
10.3 Getting the kernel run	73	16.4 Main families and classes	143
11 Useful internal definitions	74	16.5 Keyword comments	146
11.1 General purpose macros	74	16.6 Export of identifiers	148
11.2 Character tables manipulated	75	17 More aspects and keys	149
Implementation	77	17.1 Styles and languages	149
12 Overture	77	17.2 Format definitions*	151
13 General problems	80	17.3 Line numbers	157
13.1 Substring tests	80	17.4 Line shape and line breaking	160
13.2 Flow of control	83	17.5 Frames	162
13.3 Catcode changes	84	17.6 Macro use for make	170
13.4 Applications to 13.3	86	18 Typesetting a listing	171
13.5 Driver file handling*	87	18.1 Floats, boxes and captions	172
13.6 Aspect commands	90	18.2 Init and EOL	176
13.7 Interfacing with keyval	92	18.3 List of listings	180
13.8 Internal modes	93	18.4 Inline listings	181
13.9 Divers helpers	95	18.5 The input command	182
14 Doing output	96	18.6 The environment	184
14.1 Basic registers and keys	96	18.6.1 Low-level processing	184
14.2 Low- and mid-level output	98	18.6.2 \lstnewenvironment	186
14.3 Column formats	101	19 Documentation support	188
14.4 New lines	102	19.1 Required packages	189
14.5 High-level output	103	19.2 Environments for notes	189
14.6 Dropping the whole output	105	19.3 Extensions to doc	191
14.7 Writing to an external file	105	19.4 The lstsample environment	192
15 Character classes	106	19.5 Miscellaneous	193
15.1 Letters, digits and others	107	19.6 Scanning languages	196
15.2 Whitespaces	107	19.7 Bubble sort	198
15.3 Character tables	110	20 Interfaces to other programs	199
15.3.1 The standard table	110	20.1 0.21 compatibility	199
15.3.2 National characters	114	20.2 fancyvrb	201
15.3.3 Catcode problems	115	20.3 Omega support	204
15.3.4 Adjusting the table	116	20.4 LGrind	204
15.4 Delimiters	118	20.5 hyperref	208
15.4.1 Strings	124	21 Epilogue	208
15.4.2 Comments	126	22 History	209
15.4.3 PODs	128		
15.4.4 Tags	129		
15.5 Replacing input	130		
15.6 Escaping to L ^A T _E X	131		

Preface

Reading this manual If you are experienced with the listings package, you should read the next paragraph “*News and changes*”. Otherwise read section [1 Getting started](#) step by step and go on with section [2](#).

News and changes This is the second bugfix release; only few keys have been added, refer the table—note also the new section [5.7](#) describing a new experimental feature, which might change in future.

<i>new to 1.2</i>			
breakatwhitespace (page 33)			
linerrange (page 27)			
<i>1.0</i>	<i>1.1 and later</i>	<i>1.0</i>	<i>1.1 and later</i>
keywordsinside	tag (page 42)	—	final* (p. 11)
—	tagstyle (p. 28)	—	numberfirstline (p. 31)
—	markfirstintag (p. 28)	—	upquote (p. 30)
usekeywordsinside	usekeywordsintag (p. 28)	—	fvcmdparams (p. 39)
—		—	morefvcmdparams (p. 39)
<i>0.21</i>	<i>1.x</i>	<i>0.21</i>	<i>1.x</i>
first	firstline	—	numbers
last	lastline	labelstep	stepnumber
stringspaces	showstringspaces	labelstyle	numberstyle
visiblespaces	showspaces	\thelstlabel	\thelstnumber
visibletabs	showtabs	labelsep	numbersep
framerulewidth	framerule	firstlabel	firstnumber
framerulesep	rulesep	advancelabel	—
frametextsep	framesep	spread	—
framespread	superceded by	indent	xleftmargin ³
	framexleftmargin	—	xrightmargin
	framexrightmargin	wholeline	resetmargins
	framextopmargin	defaultclass	classoffset
	framexbottommargin	stringtest	—
framerulecolor	rulecolor ¹	outputpos	—
—	columns ²		

* This is an option for package loading only.

¹ All color-keys require now an explicit \color command in the value.

² Please look at section [2.10](#).

³ Now frames are also moved!

Since 2003/08/13 the following languages have been added: **bash**, **sh**, **Oz**, **Rexx**, **Inform**, **Ant**, **XSLT**. Thanks go to the contributors.

Thanks There are many people I have to thank for fruitful communication, posting their ideas, giving error reports, adding programming languages to **lstdrvs.dtx**, and so on. Their names are listed in section [3.4](#).

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

User's guide

1 Getting started

1.1 A minimal file

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

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

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

% Insert Pascal examples here.

\end{document}
```

Now type in this first example and run it through L^AT_EX.

- Must I do that really? Yes and no. Some books about programming say this is good. What a mistake! Typing takes time—wasted if the code is clear to you. And if you need that time to understand what is going on, the author of the book should reconsider the concept of presenting the crucial things—you might want to say that about this guide even—or you're simply unexperienced with programming. If only the latter case applies, you should spend more time on reading (good) books about programming, (good) documentations, and (good) source code from other people. Of course you should also make your own experiments. You will learn a lot. However, running the example through L^AT_EX shows whether the listings package is installed.
- The example doesn't work. Are the two packages listings and keyval installed on your system? Consult the administration tool of your T_EX distribution, your system administrator, the local T_EX and L^AT_EX guides, a T_EX FAQ, and section 4.18—in the given order. If you've checked *all* these sources and are still helpless, you might want to write a post to a T_EX newsgroup like comp.text.tex.
- Should I read the software license before using the package? Yes, but read this *Getting started* section first to decide whether you are willing to use the package.

1.2 Typesetting listings

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

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

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

Displayed code The `lstlisting` environment typesets the enclosed source code. Like most examples, the following one shows verbatim L^AT_EX 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 L^AT_EX.

<pre> for i:=maxint to 0 do begin { do nothing } end; Write('Case_insensitive_'); WriteE('Pascal_keywords.');</pre>	<pre> \begin{lstlisting} for i:=maxint to 0 do begin { do nothing } end; Write('Case insensitive '); WriteE('Pascal keywords. '); \end{lstlisting}</pre>
--	---

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:

<div style="border: 1px solid black; padding: 5px; width: fit-content;"> <pre> begin { do nothing } end;</pre> </div>	<pre> \begin{lstlisting}[firstline=2, lastline=5] for i:=maxint to 0 do begin { do nothing } end; Write('Case insensitive '); WriteE('Pascal keywords. '); \end{lstlisting}</pre>
---	--

- Hold on! Where comes the frame from and what is it good for? You can put frames around all listings except code snippets. You will learn it later. The frame shows that empty lines at the end of listings aren't printed. This is line 5 in the example.
- Hey, you can't drop my empty lines! You can tell the package not to drop them: The key 'showlines' controls these empty lines and is described in section 4.2. Warning: First read ahead on how to use keys in general.
- I get obscure error messages when using 'firstline'. That shouldn't happen. Make a bug report as described in section 7 *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 ');
WriteE('Pascal keywords. ');
\end{lstlisting}
```

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

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

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

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

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

Write('Case_insensitive_');
WriteE('Pascal_keywords.');
```

1.4 Seduce to use

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

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

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

	<code>\begin{lstlisting}</code>
	<code>for i:=maxint to 0 do</code>
2 <code>begin</code>	<code>begin</code>
<code>{ do nothing }</code>	<code>{ do nothing }</code>
4 <code>end;</code>	<code>end;</code>
6 <code>Write('Case_insensitive_');</code>	<code>Write('Case insensitive ');</code>
<code>WriteE('Pascal_keywords.');</code>	<code>WriteE('Pascal keywords.');</code>
	<code>\end{lstlisting}</code>

→ 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 L^AT_EX, please don't wonder: you'll miss the horizontal rules since they are described elsewhere.

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

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

	<code>\begin{lstlisting}</code>
<code>if (i≤0) then i ← 1;</code>	<code>if (i<=0) then i := 1;</code>
<code>if (i≥0) then i ← 0;</code>	<code>if (i>=0) then i := 0;</code>
<code>if (i≠0) then i ← 0;</code>	<code>if (i<>0) then i := 0;</code>
	<code>\end{lstlisting}</code>

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

1.5 Alternatives

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

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

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

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

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


```

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

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

LGrind not installed.

```

%]
```

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

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

cv2ltx is a family of 'source code to \LaTeX ' converters for C, Objective C, C++, IDL and Perl. Different styles, line numbers and other qualifiers can be chosen by command-line option. Unfortunately it isn't documented how other programming languages can be added.

C++ \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.

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

tiny_c2ltx is a C/C++/Java to \LaTeX converter based on cv2ltx (or the other way round?). It supports line numbers, block comments, \LaTeX code in/as comments, and smart line breaking. Font selection and tabulators are hard-coded, i.e. you have to rebuild the program if you want to change the appearance.

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

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

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

if $i \leq 0$ then	<code>\begin{algorithmic}</code>
$i \leftarrow 1$	<code>\IF{\$i \leq 0\$}</code>
else	<code>\STATE \$i \gets 1\$</code>
if $i \geq 0$ then	<code>\ELSE\IF{\$i \geq 0\$}</code>
$i \leftarrow 0$	<code>\STATE \$i \gets 0\$</code>
end if	<code>\ENDIF\ENDIF</code>
end if	<code>\end{algorithmic}</code>

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.

pretprn 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 automata and formal languages.

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

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

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

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

2 The next steps

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

2.1 Software license

The files `listings.dtx` and `listings.ins` and all files generated from only these two files are referred to as ‘the `listings` package’ or simply ‘the package’. A ‘driver’ is generated from `lstdrvrs.dtx`.

Copyright The `listings` package is copyright 1996–2004 Carsten Heinz. The drivers are copyright any individual author listed in the driver files.

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

Modification advice Permission is granted to modify the `listings` package as well as `lstdrvrs.dtx`. You are not allowed to distribute a modified version of the `listings` package or `lstdrvrs.dtx` unless you change the file names *and* provide the original files. In any case it is better to contact the address below; other users will welcome removed bugs, new features, and additional programming languages.

At your option To support further development, you might want to send me a copy of your document, relevant parts of it, or the crucial L^AT_EX passages. Of course, I don't need examples of normal usage.

If you distribute the package as part of a commercial product or if you use the package to prepare a commercial document (books, journals, and so on), I'd like to encourage you to make a donation to the L^AT_EX3 fund. For more information about L^AT_EX3 see <http://www.latex-project.org>.

Contacts Read section 7 *Troubleshooting* on how to submit a bug report. Send all other comments, ideas, and additional programming languages to *Carsten Heinz, Tellweg 6, 42275 Wuppertal, Germany* or preferably to cheinz@gmx.de using `listings` as part of the subject.

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

2.2 Package loading

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

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

0.21

compiles a document created with version 0.21.

draft

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

final

Overwrites a global `draft` option.

savemem

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

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

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

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

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

Table 1 on page 13 shows all defined languages and their dialects.

2.3 The key=value interface

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

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

- So I can write `\lstset{firstline=2,lastline=5}` once for all? No. ‘`firstline`’ and ‘`lastline`’ belong to a small set of keys which are used on individual listings. However, your command is not illegal—it has no effect. You have to use these keys inside the optional argument of the environment or input command.
- What’s about a better example of a key=value list? There is one in section 1.3.
- `language=[77]Fortran` does not work inside an optional argument. You must put braces around the value if a value with optional argument is used inside an optional argument. In the case here write `language={ [77]Fortran }` to select Fortran 77.
- If I use the ‘`language`’ key inside an optional argument, the language isn’t active when I typeset the next listing. All parameters set via ‘`\lstset`’ keep their values up to the end of the current environment or group. Afterwards the previous values are restored. The optional parameters of the two pretty-printing commands and the ‘`lstlisting`’ environment take effect on the particular listing only, i.e. values are restored immediately. For example, you can select a main language and change it for special listings.
- `\lstinline` has an optional argument? Yes. And from this fact comes a limitation: you can’t use the left bracket ‘`[`’ as delimiter except you specify at least an empty optional argument as in `\lstinline[] [var i:integer;]`. If you forget this, you will either get a “runaway argument” error from T_EX, or an error message from the `keyval` package.

2.4 Programming languages

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

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

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

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

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

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

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

2.5 Special characters

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

<pre>123456789 { one tabulator } { two tabs } 123 { 123 + two tabs }</pre>	<pre>\lstset{tabsize=2} \begin{lstlisting} 123456789 { one tabulator } { two tabs } 123 { 123 + two tabs } \end{lstlisting}</pre>
--	---

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:

<pre>_____for i:=maxint to 0 do _____begin _____→ { do nothing } _____end;</pre>	<pre>\lstset{showspaces=true, showtabs=true, tab=\rightarrowfill} \begin{lstlisting} for i:=maxint to 0 do begin { do nothing } end; \end{lstlisting}</pre>
--	---

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

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

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

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

→ I have problems using `inputenc` together with listings. This could be a compatibility problem. Make a bug report as described in section [7 Troubleshooting](#).

The extended characters don’t cover Arabic, Chinese, Hebrew, Japanese, and so on. Read section [8](#) for details on work-arounds.

2.6 Line numbers

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

<pre> 100 for i:=maxint to 0 do begin 102 { do nothing } end; </pre>	<pre> \begin{lstlisting}[firstnumber=100] for i:=maxint to 0 do begin { do nothing } end; \end{lstlisting} </pre>
<p>And we continue the listing:</p> <pre> 105 Write('Case_insensitive_'); WriteE('Pascal_keywords. '); </pre>	<pre> And we continue the listing: \begin{lstlisting}[firstnumber=last] Write('Case insensitive '); WriteE('Pascal keywords. '); \end{lstlisting} </pre>

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

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

<pre> for i:=maxint to 0 do 2 begin { do nothing } 4 end; </pre>	<pre> \begin{lstlisting}[name=Test] for i:=maxint to 0 do begin { do nothing } end; \end{lstlisting} </pre>
<p>And we continue the listing:</p> <pre> 6 Write('Case_insensitive_'); WriteE('Pascal_keywords. '); </pre>	<pre> And we continue the listing: \begin{lstlisting}[name=Test] Write('Case insensitive '); WriteE('Pascal keywords. '); \end{lstlisting} </pre>

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

→ Okay. And how can I get decreasing line numbers? Sorry, what? Decreasing line numbers as on page 32. May I suggest to demonstrate your individuality by other means? If you differ, you should try a negative 'stepnumber' (together with '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.

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

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

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

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

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

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

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

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

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

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

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


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

- Don't use frames all the time, in particular not with short listings. This would emphasize nothing. Use frames for 10% or even less of your listings, for your most important ones.
- If you use frames on floating listings, do you really want frames? No, I want to separate floats from text. Then it is better to redefine L^AT_EX's '`\topfigrule`' and '`\botfigrule`'. For example, you could write '`\renewcommand*\topfigrule{\hrule\kern-0.4pt\relax}`' and make the same definition for `\botfigrule`.

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

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

Listing 2: Useless code

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

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

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

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

'Caption' without label

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

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

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

→ Great! I love colours. Fine, yes, really. And I like to remind you of the warning about striking styles on page 6.

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

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

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

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

2.8 Emphasize identifiers

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

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

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

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

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

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

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

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

Both keys have an optional *<class number>* argument for multiple identifier lists:

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

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

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

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

2.9 Indexing

is just like emphasizing identifiers—I mean the usage:

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

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

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

- Why the 'index' key is able to work with multiple identifier lists? This question is strongly related to the 'indexstyle' key. Someone might want to create multiple indexes or want to insert prefixes like 'constants', 'functions', 'keywords', and so on. The 'indexstyle' key works like the other style keys except that the last token *must* take an argument, namely the (printable form of the) current identifier. You can define '\newcommand\indexkeywords[1]{\index{keywords, #1}}' and make similar definitions for constant or function names. Then 'indexstyle=[1]\indexkeywords' might meet your purpose. This becomes easier if you want to create multiple indexes with the **index** package. If you have defined appropriate new indexes, it is possible to write 'indexstyle=\index[keywords]', for example.
- Let's say, I want to index all keywords. It would be annoying to type in all the keywords again, specifically if the used programming language changes frequently. Just read ahead.

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

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

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

2.10 Fixed and flexible columns

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

<code>columns=</code>	<code>fixed</code> (at 0.6em)	<code>flexible</code> (at 0.45em)	<code>fullflexible</code> (at 0.45em)
WOMEN are	WOMEN are	WOMEN are	WOMEN are
MEN	MEN	MEN	MEN
WOMEN are	WOMEN are	WOMEN are	WOMEN are
better MEN	better MEN	better MEN	better MEN

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

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

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

Now, the fixed format puts n characters into a box of width $n \times$ ‘base width’, where the base width is 0.6em in the example. The format shrinks and stretches the space between the characters to make them fit the box. As shown in the example, some character strings look 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 two formats is that the full flexible format cares about nothing else and the normal flexible format tries to fix the column alignment if a character string needs less space than ‘reserved’. In the flexible example above, the two MENs are vertically aligned since some space has been inserted in the fourth line to fix the alignment. In the full flexible format, the two MENs are not aligned.

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

3 Advanced techniques

3.1 Style definitions

It is obvious that a pretty-printing tool like this requires some kind of language selection and definition. The first has already been described and the latter is 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, `style=<style name>` activates a previously defined style. A definition is as easy: `\lstdefinestyle{<style name>}{<key=value list>}`. Keys not used in such a definition are untouched by the corresponding style selection, of course. For example, you could write

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

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

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

3.2 Language definitions

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

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

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

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

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

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

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

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

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

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

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

3.3 Delimiters

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

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

// bold comment line
a single

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

As you can see, you have the choice between specifying the style explicitly by L^AT_EX 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: `keywordstyle2`.

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

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

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

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

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

- Please remember that both 'extra' comments must be defined *after* the normal comment, since the delimiter `/*` is a substring of `/*+` and `/*-`.
- I have another question. Is `'language=(different language)'` the only way to remove such additional delimiters? Call `deletecomment` and/or `deletestring` with the same arguments to remove the delimiters (but you don't need to provide the optional style argument).

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

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

begin end
beginend

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

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

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

roman typewriter

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

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

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

/* begin
(* comment *)
' _string_ ' */

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

Moreover, you can force the styles being applied cumulative.

```

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

/* begin
   '\string_'
   typewriter */

begin
'\string_'
/*typewriter*/

\begin{lstlisting}
/* begin
   ' string '
   |typewriter| */

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

```

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

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

3.4 Closing and credits

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

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

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

Moreover I wish to thank

Bjørn Ådlandsvik, Gaurav Aggarwal, Jason Alexander, Donald Arseneau,
David Aspinall, Claus Atzenbeck, Peter Bartke (big thankyou),
Oliver Baum, Ralph Becket, Andres Becerra Sandoval, Javier Bezos,
Olaf Trygve Berglihn, Geraint Paul Bevan, Peter Biechele, Kai Below,
Beat Birkhofer, Frédéric Boulanger, Martin Brodbeck, Walter E. Brown,
Achim D. Brucker, David Carlisle, Bradford Chamberlain, Patrick Cousot,
Xavier Crégut, Holger Danielsson, Andreas Deininger, Robert Denham,
Detlev Dröge, Anders Edenbrandt, Mark van Eijk, Norbert Eisinger,
Thomas Esser, Chris Edwards, David John Evans, Tanguy Fautré, Robert Frank,
Daniel Gazard, Daniel Gerigk, KP Gores, Adam Grabowski, Jean-Philippe Grivet,
Christian Gudrian, Jonathan de Halleux, Carsten Hamm, Martina Hansel,
Harald Harders, Christian Haul, Aidan Philip Heerdegen, Jim Hefferon,
Heiko Heil, Jürgen Heim, Alvaro Herrera, Dr. Jobst Hoffmann,
Torben Hoffmann, Richard Hoefter, Berthold Höllmann, Hermann Hüttler,
Ralf Imhäuser, R. Isernhagen, Oldrich Jedlicka, Dirk Jesko, Christian Kaiser,
Bekir Karaoglu, Marcin Kasperski, Christian Kindinger, Steffen Klupsch,
Peter Köller (big thankyou), Stefan Lagotzki, Tino Langer, Rene H. Larsen,
Olivier Lecarme, Thomas Leduc, Dr. Peter Leibner, Thomas Leonhardt
(big thankyou), Magnus Lewis-Smith, Knut Lickert, Dan Luecking,
Kris Luyten, José Romildo Malaquias, Andreas Matthias, Riccardo Murri,
Knut Müller, Svend Tollak Munkejord, Gerd Neugebauer, Torsten Neuer,
Patrick T.J. McPhee, Michael Niedermair, Xavier Noria, Heiko Oberdiek,
Markus Pahlow, Morten H. Pedersen, Zvezdan V. Petkovic, Michael Piefel,

Michael Piotrowski, Manfred Piringer, Vincent Poirriez, Adam Prugel-Bennett, Ralf Quast, Aslak Raanes, Venkatesh Prasad Ranganath, Georg Rehm, Fermin Reig, Detlef Reimers, Stephen Reindl, Peter Ruckdeschel, Magne Rudshaug, Jonathan Sauer, Vespe Savikko, Gunther Schmidl, Walter Schmidt, Jochen Schneider, Benjamin Schubert, Uwe Siart, Axel Sommerfeldt, Richard Stallman, Martin Steffen, Andreas Stephan, Stefan Stoll, Enrico Straube, Werner Struckmann, Martin Süßkraut, Gabriel Tauro, Winfried Theis, Jens T. Berger Thielemann, Arnaud Tisserand, Kalle Tuulos, Gregory Van Vooren, Thorsten Vitt, Herbert Voss (big thankyou), Herfried Karl Wagner, Dominique de Waleffe, Jared Warren, Michael Weber, Sonja Weidmann, Herbert Weinhandl, Robert Wenner, Michael Wiese, Jörn Wilms, Kai Wollenweber, Ulrich G. Wortmann, Timothy Van Zandt, and Edsko de Vries.

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

Reference guide

4 Main reference

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

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

4.1 How to read the reference

Commands, keys and environments are presented as follows.

hints **command**, **environment** or **key** with *(parameters)* **default**

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

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

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

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

Regarding the parameters, please keep in mind the following:

1. A list always means a comma separated list. You must put braces around such a list. Otherwise you'll get in trouble with the `keyval` package; it complains about an undefined key.
2. You must put parameter braces around the whole value of a key if you use an `[optional argument]` of a key inside an optional `[key=value list]`: `\begin{lstlisting}[caption={[one]two}]`.

3. Brackets ‘[]’ usually enclose optional arguments and must be typed in verbatim. Normal brackets ‘[]’ always indicate an optional argument and must not be typed in. Thus [∗] must be typed in exactly as is, but [∗] just gets ∗ if you use this argument.
4. A vertical rule indicates an alternative, e.g. <true|false> allows either true or false as arguments.
5. If you want to enter one of the special characters {}#%\, this character must be escaped with a backslash. This means that you must write \} for the single character ‘right brace’—but of course not for the closing 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 opposition, all optional <key=value list>s below modify the parameters for single listings only.

`\lstinline[<key=value list>]<character><source code><same character>`

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

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

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

`\begin{lstlisting}[<key=value list>]`

`\end{lstlisting}`

typesets the code in between as a displayed listing.

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

`\lstinputlisting[<key=value list>]{<file name>}`

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

4.3 Space and placement

`float=[∗]<subset of tbph>` or `float` `floatplacement`

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

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

`floatplacement`= $\langle place\ specifiers \rangle$ tbp
 is used as place specifier if `float` is used without value.

`aboveskip`= $\langle dimension \rangle$ \medskipamount
`belowskip`= $\langle dimension \rangle$ \medskipamount
 define the space above and below displayed listings.

\dagger `lineskip`= $\langle dimension \rangle$ Opt
 specifies additional space between lines in listings.

\dagger `boxpos`= $\langle b|c|t \rangle$ c
 Sometimes the `listings` package puts a `\hbox` around a listing—or it couldn't be printed or even processed correctly. The key determines the vertical alignment to the surrounding material: bottom baseline, centered or top baseline.

4.4 The printed range

`print`= $\langle true|false \rangle$ or `print` true
 controls whether an individual displayed listing is typeset. Even if set false, the respective caption is printed and the label is defined.

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

`firstline`= $\langle number \rangle$ 1
`lastline`= $\langle number \rangle$ 9999999
 can be used on individual listings only. They determine the physical input lines used to print displayed listings.

`linerrange`= $\{ \langle first1 \rangle - \langle last1 \rangle, \langle first2 \rangle - \langle last2 \rangle, \text{ and so on } \}$ 1.2
 can be used on individual listings only. The given line ranges of the listing are displayed. The intervals must be sorted and must not intersect.

`showlines`= $\langle true|false \rangle$ or `showlines` false
 If true, the package prints empty lines at the end of listings. Otherwise these lines are dropped (but they count for line numbering).

`emptylines`= $[*] \langle number \rangle$
 sets the maximum of empty lines allowed. If there is a block of more than $\langle number \rangle$ empty lines, only $\langle number \rangle$ ones are printed. Without the optional star, line numbers can be disturb when blank lines are omitted; with the star, the lines keep their original numbers.

`gobble`= $\langle number \rangle$ 0
 gobbles $\langle number \rangle$ characters at the beginning of each *environment* code line. This key has no effect on `\lstinline` or `\lstinputlisting`.

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

4.5 Languages and styles

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

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

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

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

stores the key=value list.

language=[$\langle dialect \rangle$] $\langle language \rangle$ {}

activates a (dialect of a) programming language. The ‘empty’ default language detects no keywords, no comments, no strings, and so on. Without specifying $\langle dialect \rangle$, the package chooses a default dialect.

Table 1 on page 13 lists all languages and dialects provided by `lstdrvrs.dtx`. The predefined default dialects are underlined.

alsolanguage=[$\langle dialect \rangle$] $\langle language \rangle$

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

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

defaultdialect=[$\langle dialect \rangle$] $\langle language \rangle$

defines $\langle dialect \rangle$ as default dialect for $\langle language \rangle$. If you have defined a default dialect other than empty, for example `defaultdialect=[iama]fool`, you can’t select the empty dialect, even not with `language=[]fool`.

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

optional **print**pod= $\langle true|false \rangle$ false

prints or drops PODs in Perl.

renamed, optional **use**keywordsintag= $\langle true|false \rangle$ true

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

optional **tag**style= $\langle style \rangle$ {}

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

optional **mark**firstintag= $\langle style \rangle$ false

prints the first name in tags with keyword style.

optional **make**macrouse= $\langle true|false \rangle$ true

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

4.6 Figure out the appearance

`basicstyle=<basic style>` {}

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

`identifierstyle=<style>` {}

`commentstyle=<style>` \itshape

`stringstyle=<style>` {}

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

`keywordstyle=[<number>]<style>` \bfseries

`ndkeywordstyle=<style>` keywordstyle

are used to print keywords and second order keywords (if defined). The optional `<number>` argument is the class number to which the style should be applied. `ndkeywordstyle=...` is equivalent to `keywordstyle=[2]...`

`classoffset=<number>` 0

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

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

```

one two three
four five six

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

optional `texcsstyle=<style>` keywordstyle

optional `directivestyle=<style>` keywordstyle

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

`emph=[<number>]{<identifier list>}`

`moreemph=[<number>]{<identifier list>}`

`deleteemph=[<number>]{<identifier list>}`

`emphstyle=[<number>]{<style>}`

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

These keys are described more detailed in section 2.8.

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

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

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

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

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

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

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

4.7 Getting all characters right

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

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

`inputencoding=⟨encoding⟩` `{}`

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

`upquote=⟨true|false⟩` `false`

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

`tabsize=⟨number⟩` `8`

sets tabulator stops at columns $\langle number \rangle + 1$, $2 \cdot \langle number \rangle + 1$, $3 \cdot \langle number \rangle + 1$, 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 `_____`, but that can be changed. If you choose invisible tabulators but visible spaces, tabulators are converted to an appropriate number of spaces.

`tab=⟨tokens⟩`

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

`showspaces=<true|false>` false
lets all blank spaces appear `_` or as blank spaces.

`showstringspaces=<true|false>` true
lets blank spaces in strings appear `_` or as blank spaces.

`formfeed=<tokens>` `\bigbreak`
Whenever a listing contains a form feed `<tokens>` is executed.

4.8 Line numbers

`numbers=<none|left|right>` none
makes the package either print no line numbers, or put them on the left or the right side of a listing.

`stepnumber=<number>` 1
All lines with “line number $\equiv 0$ modulo `<number>`” get a line number. If you turn line numbers on and off with `numbers`, the parameter `stepnumber` will keep its value. Alternatively you can turn them off via `stepnumber=0` and on with a nonzero number and keep the value of `numbers`.

`numberfirstline=<true|false>` false
The first line of each listing gets numbered (if numbers are on at all) even if the line number is not divisible by `stepnumber`.

`numberstyle=<style>` {}
determines the font and size of the numbers.

`numbersep=<dimension>` 10pt
is the distance between number and listing.

`numberblanklines=<true|false>` true
If this is set to false, blank lines get no printed line number.

`firstnumber=<auto|last|<number>>` auto
`auto` lets the package choose the first number: a new listing starts with number one, a named listing continues the most recent same-named listing (see below), and a stand alone file begins with the number corresponding to the first input line.
`last` continues the numbering of the most recent listing and `<number>` sets it to the number.

`name=<name>`
names a listing. Displayed environment-listings with the same name share a line counter.

data `\thelstnumber` `\arabic{lstnumber}`
prints the lines’ numbers.

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

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

	<code>\begin{lstlisting}[numbers=left,</code>
	<code>firstnumber=753]</code>
753	<code>begin { empty lines }</code>
752	<code>begin { empty lines }</code>
751	
750	
749	
748	
747	
746	<code>end; { empty lines }</code>
	<code>end; { empty lines }</code>
	<code>\end{lstlisting}</code>

→ The example shows a sequence $n, n+1, \dots, n+7$ of 8 three-digit figures such that the sequence contains each digit $0, 1, \dots, 9$. 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 $k \in \{1, \dots, 10\}$ be given. Find the minimal number $m \in \{1, \dots, 10\}$ such that there is a sequence $n, n+1, \dots, n+m-1$ of m k -digit figures which contains each digit $\{0, \dots, 9\}$. Prove that the number is minimal. How many minimal sequences do exist? If you solve this problem with a computer, write a \TeX program!

4.9 Captions

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

`title=<title text>`

is used for a title without any numbering or label.

`caption={[[<short>]]<caption text>}`

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

`label=<name>`

makes a listing referable via `\ref{<name>}`.

`\lstlistoflistings`

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

`nolol={true|false}` or `nolol`

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

data `\lstlistlistingname`

Listings

The header name for the list of listings.

data `\lstlistingname` Listing

The caption label for listings.

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

prints the running number of the caption.

`\lstname`

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

`captionpos`=*<subset of tb>* t

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

`abovecaptionskip`=*<dimension>* `\smallskipamount`

`belowcaptionskip`=*<dimension>* `\smallskipamount`

is the vertical space above respectively below each caption.

4.10 Margins and line shape

`linewidth`=*<dimension>* `\linewidth`

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

`xleftmargin`=*<dimension>* Opt

`xrightmargin`=*<dimension>* Opt

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

`resetmargins`=*<true|false>* false

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

`breaklines`=*<true|false>* or `breaklines` false

activates or deactivates automatic line breaking of long lines.

`breakatwhitespace`=*<true|false>* or `breakatwhitespace` false 1.2

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`= \langle true|false \rangle 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 indentation to zero.

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

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

      "A_long_string_
        is_broken!"
      " Another_
        long_
        line."

      { Now auto
indentation is off. }
```

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

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

4.11 Frames

`frame`= \langle none|leftline|topline|bottomline|lines|single|shadowbox \rangle none
 draws either no frame, a single line on the left, at the top, at the bottom, at the top and bottom, a whole single frame, or a shadowbox.

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

`frame`= \langle subset of trblTRBL \rangle {}

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

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

`frameround`= \langle t|f \rangle \langle t|f \rangle \langle t|f \rangle \langle t|f \rangle ffff

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

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

`framesep`= \langle dimension \rangle 3pt

`rulesep`= \langle dimension \rangle 2pt

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

`framerule`= \langle dimension \rangle 0.4pt

controls the width of the rules.

`framexleftmargin=<dimension>` Opt
`framexrightmargin=<dimension>` Opt
`framextopmargin=<dimension>` Opt
`framexbottommargin=<dimension>` Opt

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

`backgroundcolor=<color command>`
`rulecolor=<color command>`
`fillcolor=<color command>`
`rulesepcolor=<color command>`

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

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

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

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

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

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

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

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

† `frameshape={<top shape>}{<left shape>}{<right shape>}{<bottom shape>}`

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

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

`<top shape>` and `<bottom shape>` are ‘left-rule-right’ sequences (or empty). The first ‘left-rule-right’ sequence is attached to the most inner rule, the second to the next, and so on. Each sequence has three characters: ‘rule’ is

either `y` or `n`; ‘left’ and ‘right’ are `y`, `n` or `r` (which makes a corner round). The example uses `RYRYNYYYY` for both shapes: `RYR` describes the most inner (top and bottom) frame shape, `YNY` the middle, and `YYY` the most outer.

To summarize, the example above used

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

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

4.12 Indexing

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

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

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

define, add and remove $\langle identifiers \rangle$ and $\langle keyword\ classes \rangle$ 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=[⟨number⟩] ⟨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.

4.13 Column alignment

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

selects the respective column format, refer section 2.10.

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

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

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

`† keepspaces=<true|false>` false

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

`basewidth=<dimension>` or

`basewidth={<fixed>,<flexible mode>}` {0.6em,0.45em}

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

`fontadjust=<true|false>` or `fontadjust` false

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

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

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

<pre> { scriptsize font doesn't look good } for i:=maxint to 0 do begin { do nothing } end; </pre>	<pre> \lstset{commentstyle=\scriptsize} \begin{lstlisting} { scriptsize font doesn't look good } for i:=maxint to 0 do begin { do nothing } end; \end{lstlisting} </pre>
<pre> { scriptsize font looks better now } for i:=maxint to 0 do begin { do nothing } end; </pre>	<pre> \begin{lstlisting}[fontadjust] { scriptsize font looks better now } for i:=maxint to 0 do begin { do nothing } end; \end{lstlisting} </pre>

4.14 Escaping to L^AT_EX

Note: Any escape to L^AT_EX may disturb the column alignment since the package can’t control the spacing there.

`texcl=<true|false>` or `texcl` false

activates or deactivates L^AT_EX 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 L^AT_EX code and typeset in comment style.

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

```

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

```

`mathescape=(true|false)` false

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

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

`escapechar=(character)` or `escapechar={}` {}

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

`escapeinside=(character)(character)` or `escapeinside={}` {}

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

`escapebegin=(tokens)` {}

`escapeend=(tokens)` {}

The tokens are executed at the beginning respectively at the end of each escape, in particular for `texcl`. See section 8 for an application.

<pre> // calculate a_{ij} $a_{ij} = a_{jj}/a_{ij};$ </pre>	<pre> \begin{lstlisting}[mathescape] // calculate a_{ij} $a_{ij} = a_{jj}/a_{ij};$ \end{lstlisting} </pre>
<pre> // calculate a_{ij} $a_{ij} = a_{jj}/a_{ij};$ </pre>	<pre> \begin{lstlisting}[escapechar=\%] // calculate a_{ij} $a_{ij} = a_{jj}/a_{ij};$ \end{lstlisting} </pre>
<pre> // calculate a_{ij} $a_{ij} = a_{jj}/a_{ij};$ </pre>	<pre> \lstset{escapeinside=''} \begin{lstlisting} // calculate a_{ij} $a_{ij} = a_{jj}/a_{ij};$ \end{lstlisting} </pre>

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

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

1. Don't use any command of the listings package when you have escaped to \LaTeX .
2. Any environment must start and end inside the same escape.

3. You might use `\def`, `\edef`, etc., but do not assume that the definitions are present later—except they are `\global`.
4. `\if \else \fi`, groups, math shifts `$` and `$$`, ... must be balanced each escape.
5. ...

Expand that list yourself and mail me about new items.

4.15 Interface to `fancyvrb`

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

`fancyvrb`= \langle true|false \rangle

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

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

`fvcmdparams`= \langle command₁ \rangle \langle number₁ \rangle ... `\overlay1`

`morefvcmdparams`= \langle command₁ \rangle \langle number₁ \rangle ...

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

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

First verbatim line.
Second verbatim line.

```
\lstset{morecomment=[1]\ }% :-)
\fvset{commandchars=\\{\}}
```

```
\begin{BVerbatim}
First verbatim line.
\fbx{Second} verbatim line.
\end{BVerbatim}
```

```
\par\vspace{72.27pt}
```

First verbatim line.
Second verbatim line.

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

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

4.16 Environments

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

```
\lstnewenvironment
  {\langle name \rangle} [\langle number \rangle] [\langle opt. default arg. \rangle]
  {\langle starting code \rangle}
  {\langle ending code \rangle}
```

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

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

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

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

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

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

4.17 Language definitions

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

```
\lstdefinlanguage
  [[\langle dialect \rangle]] {\langle language \rangle}
  [[\langle base dialect \rangle]] {\langle and base language \rangle}
  {\langle key=value list \rangle}
  [[\langle list of required aspects (keywordcomments,txcs,etc.) \rangle]]
```

defines the (given dialect of the) programming language `\langle language \rangle`. If the language definition is based on another definition, you must specify the whole `[[\langle base dialect \rangle]] {\langle and base language \rangle}`. Note that an empty `\langle base dialect \rangle` 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@definlanguage` has the same syntax and is used to define languages in the driver files.

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

`\lstalias{⟨alias⟩}{⟨language⟩}`

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

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

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

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

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

†bug `keywordsprefix=⟨prefix⟩`

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

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

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

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

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

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

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

`ndkeywords={⟨list of keywords⟩}`

`morendkeywords={⟨list of keywords⟩}`

`deletendkeywords={⟨list of keywords⟩}`

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

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

optional **moretexcs**={\<list of control sequences (without backslashes)>}

optional **deletetexcs**={\<list of control sequences (without backslashes)>}

Ditto for control sequences in T_EX and L^AT_EX.

optional **directives**={\<list of compiler directives>}

optional **moredirectives**={\<list of compiler directives>}

optional **deletedirectives**={\<list of compiler directives>}

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

sensitive=\<true|false>

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

alsoletter={\<character sequence>}

alsodigit={\<character sequence>}

alsoother={\<character sequence>}

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

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

otherkeywords={\<keywords>}

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

renamed, optional **tag**=\<character>\<character> or **tag**={\}

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

Strings

string=[\<b|d|m|bd>]{\<delimiter (character)>}

morestring=[\<b|d|m|bd>]{\<delimiter>}

deletestring=[\<b|d|m|bd>]{\<delimiter>}

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

Table 2: Standard character table

class	characters
letter	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 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 @ \$ _
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.

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

Comments

`comment`=[*<type>*]*<delimiter(s)>*

`morecomment`=[*<type>*]*<delimiter(s)>*

`deletecomment`=[*<type>*]*<delimiter(s)>*

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

`morecomment`=[*l*]*<delimiter>*

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

`morecomment`=[*s*]{*<delimiter>*}{*<delimiter>*}

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

`morecomment`=[*n*]{*<delimiter>*}{*<delimiter>*}

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

`morecomment=[f] [] [$\langle n=\textit{preceding columns} \rangle$] $\langle \textit{delimiter} \rangle$`

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={ $\langle \textit{keywords} \rangle$ }`

optional `morekeywordcomment={ $\langle \textit{keywords} \rangle$ }`

optional `deletekeywordcomment={ $\langle \textit{keywords} \rangle$ }`

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={ $\langle \textit{keywords} \rangle$ }{ $\langle \textit{keywords} \rangle$ }{ $\langle \textit{keywords} \rangle$ }`

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

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

optional `podcomment= $\langle \text{true|false} \rangle$`

activates or deactivates PODs—Perl specific.

4.18 Installation

Software installation

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

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

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

2. Create the directory `texmf/tex/latex/listings` or remove all files except `lst $\langle \textit{whatever} \rangle$ 0.sty` and `lstlocal.cfg` from that directory.
3. Change the working directory to `texmf/source/latex/listings` and run `listings.ins` through T_EX.
4. Move the generated files to `texmf/tex/latex/listings` if this is not already done.

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

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

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

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

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

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

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

data `\lstaspectfiles` contains `lstmisc0.sty`, `lstmisc.sty`

data `\lstlanguagefiles` contains `lstlang0.sty`, `lstlang1.sty`, `lstlang2.sty`, `lstlang3.sty`

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

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

5 Experimental features

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

5.1 Listings inside arguments

There are some things to consider if you want to use `\lstinline` or the listing environment inside arguments. Since T_EX reads the argument before the 'lst-macro' is executed, this package can't do anything to preserve the input: spaces shrink to one space, the tabulator and the end of line are converted to spaces, T_EX's comment character is not printable, and so on. Hence, *you* must work a bit

more. You have to put a backslash in front of each of the following four characters: `\{}%`. Moreover you must protect spaces in the same manner if: (i) there are two or more spaces following each other or (ii) the space is the first character in the line. That's not enough: Each line must be terminated with a 'line feed' `^^J`. And you can't escape to L^AT_EX 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
\fbbox{\lstinline!\\\{\}\%!\}}
```

yields¹ 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:

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

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

†optional `procnameskeys={⟨keywords⟩}` `{}`

†optional `moreprocnameskeys={⟨keywords⟩}`

†optional `deleteprocnameskeys={⟨keywords⟩}`

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

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

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

defines the style in which procedure and function names appear.

¹`\var i:integer;` and `protected` spaces and `\{}%`

†optional `indexprocnames`= \langle true|false \rangle false

If activated, procedure and function names are also indexed.

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

5.3 † Hyper references

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

†optional `hyperref`= $\{\langle$ identifiers $\rangle\}$

†optional `morehyperref`= $\{\langle$ identifiers $\rangle\}$

†optional `deletehyperref`= $\{\langle$ identifiers $\rangle\}$

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

†optional `hyperanchor`= \langle two-parameter macro \rangle `\hyper@@anchor`

†optional `hyperlink`= \langle two-parameter macro \rangle `\hyperlink`

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

5.4 Literate programming

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

<pre> var i:integer; if (i≤0) i ← 1; if (i≥0) i ← 0; if (i≠0) i ← 0; </pre>	<pre> \begin{lstlisting} var i:integer; if (i<=0) i := 1; if (i>=0) i := 0; if (i<>0) i := 0; \end{lstlisting} </pre>
--	--

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

† `literate`= \langle replacement item \rangle ... \langle replacement item \rangle

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

Each ‘printing unit’ in $\langle replacement\ text \rangle$ must be braced except it’s a single character. For example, you must put braces around $\$ \leq \$$. If you want to replace $\langle -1- \rangle$ by $\$ \leftarrow 1 \rightarrow \$$, the replacement item would be $\{ \langle -1- \rangle \} \{ \{ \$ \leftarrow \$ \} 1 \{ \$ \rightarrow \$ \} \} 3$. Note the braces around the arrows.

If one $\langle replace \rangle$ is a subsequence of another $\langle replace \rangle$, you must use the shorter sequence first. For example, $\{-\}$ must be used before $\{--\}$ and this before $\{-->\}$.

In the example above I’ve used

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

To do: Of course, it’s good to have keys for adding and removing single $\langle replacement\ item \rangle$ 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 .

optional **lgrindef**= $\langle language \rangle$

scans the **lgrindef** language definition file for $\langle language \rangle$ and activates it if present. Note that not all LGrind capabilities have a **listings** analogue.

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

data, optional **\lstlgrindef***file* **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

$$\langle input\ chars \rangle = [\langle exceptional\ chars \rangle] \langle pre \rangle [\langle \backslash string \rangle] \langle post \rangle$$

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

- $\backslash newline$ —ensuring a new line;
- $\backslash space$ —ensuring a whitespace;

- `\indent` —increasing indentation;
- `\noindent` —decreasing indentation.

Now we can give an example.

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

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

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

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

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

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

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

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

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

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

5.7 Arbitrary linerange markers

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

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

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

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

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

`rangebeginprefix`=*prefix* 1.2

`rangebeginsuffix`=*suffix* 1.2

`rangeendprefix`=*prefix* 1.2

`rangeendsuffix`=*suffix* 1.2

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

`rangeprefix`=*prefix* 1.2

`rangesuffix`=*suffix* 1.2

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

`includerangemarker`=*true|false* true 1.2

shows or hides the markers in the output.

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

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

6 Forthcoming ?

This section is rather rudimentary. It just lists some things I don't want to forget.

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

Then, there are several ideas for the future. Some have already been stated as 'to do's; some came from other people and are stated below; some more are far from being implemented, e.g. `linerange=[inter]{line range list}` which prints all lines in the range and executes *inter* when omitting some code lines. The main problem here are frames and background colours; what should happen to them? In fact, the problem is how this can be coded. Another idea is to change the background colour (or the basic style) for particular code blocks. This, too, is not easy.

Vincent Poirriez: Inside caml comments, [and] should print the code in between in basicstyle (or another newly introduced style). Nesting of these 'code example delimiters' is allowed, e.g. `(* [[x;y]] *)`.

Claus Atzenbeck: issue warning in final mode if `extendedchars=false` but extended chars are used.

Andreas Matthias: Make the header/footer print the listing name. Some people asked for continued captions.

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 ...

7 Troubleshooting

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

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

8 How tos

How to reference line numbers

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

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

```

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

for i:=maxint to 0 do
begin
  { comment }
end;
Line 3 shows a comment.
\begin{lstlisting}
for i:=maxint to 0 do
begin
  { comment }(*@\label{comment}@*)
end;
\end{lstlisting}
Line \ref{comment} shows a comment.
```

- Can I use '(*@' and '@*)' instead? Yes.
- Can I use '(*' and '@*)' instead? Sure. If you want this.
- Can I use '{@' and '@}' instead? No, never! The second delimiter is not allowed. The character '@' is defined to check whether the escape is over. But reading the lonely 'end-argument' brace, `TEX` encounters the error 'Argument of @ has an extra }'. Sorry.
- Can I use '{' and '}' instead? No. Again the second delimiter is not allowed. Here now `TEX` would give you a 'Runaway argument' error. Since '}' is defined to check whether the escape is over, it won't work as 'end-argument' brace.
- And how can I use a comment line? For example, write 'escapeinside={/*}{\^M}'. Here `\^M` represents the end of line character.

How to gobble characters

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

```

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

Write('Case_insensitive_');
Write('Pascal_keywords.');
```

```

\begin{lstlisting}[gobble=3]
1_ _for i:=maxint_ to_ _0_ do
2_ _begin
3_ _ _ _ _ {do_ nothing_}
123end;

_ _ _Write('Case_ insensitive_');
_ _ _Write('Pascal_keywords.');
```

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

- Could I use ‘gobble’ together with ‘\lstinputlisting’? Yes, but it has no effect.
- Note that ‘gobble’ can also be set via ‘\lstset’.

How to include graphics

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

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

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

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

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

How to get closed frames on each page

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

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

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

How to print national characters with Λ and listings

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

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

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

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

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

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

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

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

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

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

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

How to get bold typewriter type keywords

Use the [LuxiMono](#) package.

How to get the developer's guide

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

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

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

Developer's guide

First I must apologize for this developer's guide since some parts are not explained as good as possible. But note that you are in a pretty good shape: this developer's guide exists! You might want to peak 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:

option `noaspects`

deletes the list of aspects to load. Note that, for example, the option lists `0.21,!labels,noaspects` and `noaspects` are essentially the same: the kernel is loaded and no other aspect.

This is especially useful for aspect-testing since we can load exactly the required parts. Note, however, that an aspect is loaded later if a predefined programming language requests it. One can load aspects also by hand:

`\lstloadaspects{comma separated list of aspect names}`

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

`string, morestring, deletestring, stringstyle, showstringspaces`

comments

`comment, morecomment, deletecomment, commentstyle`

pod

`printpod, podcomment`

escape

`texcl, escapebegin, escapeend, escapechar, escapeinside, mathescape`

`writetofile` requires 1 `\toks`, 1 `\write`
`\lst@BeginWriteFile`, `\lst@BeginAlsoWriteFile`, `\lst@EndWriteFile`

`style`
`empty style`, `style`, `\lstdefinestyle`, `\lst@definestyle`,
`\lststylefiles`

`language`
`empty language`, `language`, `also language`, `defaultdialect`, `\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`,
`keywordcommentsemicolon`

`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, framextopmargin,
framexbottommargin, backgroundcolor, fillcolor, rulecolor,
rulesepcolor, rulesep, framerule, framesep, frameshape, frameround,
frame

make requires keywords

makemacrouse

doc requires writefile and 1 \box

lstsample, lstxsample

0.21 defines old keys in terms of the new ones.

fancyvrb requires 1 \box

fancyvrb, fvcmdparams, morefvcmdparams

lgrind

lgrindef, \lstlgrindeffile

hyper requires keywords

hyperref, morehyperref, deletehyperref, hyperanchor, hyperlink

The kernel allocates 6 \count, 4 \dimen and 1 \toks. Moreover it defines the following keys, commands, and environments:

basewidth, fontadjust, columns, flexiblecolumns, identifierstyle,
tabsize, showtabs, tab, showspaces, keepspaces, formfeed,
SelectCharTable, MoreSelectCharTable, extendedchars, alsoletter,
alsodigit, alsoother, excludedelims, literate, basicstyle,
print, firstline, lastline, linerange, nolol, captionpos,
abovecaptionskip, belowcaptionskip, label, title, caption,
\lstlistingname, boxpos, float, floatplacement, aboveskip,
belowskip, everydisplay, showlines, emptylines, gobble, name,
\lstname, \lstlistlistingname, \lstlistoflistings,
\lstnewenvironment, \lstinline, \lstinputlisting, lstlisting,
\lstloadaspects, \lstset, \thelstlisting, \lstaspectfiles,
inputencoding, delim, moredelim, deletedelim, upquote, fancyvrb

9.2 How to define 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 `\lst@BeginAspect` plus arguments and ends with the next `\lst@EndAspect`. In particular, aspect definitions can't be nested.

```
\lst@BeginAspect[[(list of required aspects)]]{(aspect name)}
```

```
\lst@EndAspect
```

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. $\langle \textit{aspect name} \rangle$ is empty.
2. The aspect is already defined.
3. A required aspect is neither defined nor loadable via `\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. $\langle \textit{definition} \rangle$ is the replacement text of a macro with one parameter. The argument is either the value from 'key=value' or $\langle \textit{default value} \rangle$ if no '=value' is given. The helper macros `\lstKV@...` below might simplify $\langle \textit{definition} \rangle$.

The key is not initialized if the second argument is `\relax`. Otherwise $\langle \textit{init value} \rangle$ is the initial value given to the key. Note that we locally switch to `\globaldefs=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 $\langle \text{\TeX material} \rangle$ 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 `\globaldefs` 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@AddTo` 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 $\langle definition \rangle$ argument of the `\lst@Key` command to evaluate the argument. The additional prefix KV refers to the `keyval` package.

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

$\langle if macro \rangle$ becomes `\iftrue` if the first character of $\langle value \rangle$ equals `t` or `T`. Otherwise it becomes `\iffalse`. Usually you will use `#1` as $\langle value \rangle$.

`\lstKV@SwitchCases{ $\langle value \rangle$ }`

```
{ $\langle string 1 \rangle$ & $\langle execute 1 \rangle$ \\
 $\langle string 2 \rangle$ & $\langle execute 2 \rangle$ \\
:
 $\langle string n \rangle$ & $\langle execute n \rangle$ }{ $\langle else \rangle$ }
```

Either execute $\langle else \rangle$ or the $\langle value \rangle$ matching part.

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

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

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

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

`\lstKV@OptArg[$\langle default arg. \rangle$]{ $\langle value \rangle$ }{ $\langle subdefinition \rangle$ }`

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

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

`\lstKV@XOptArg[$\langle default arg. \rangle$]{ $\langle value \rangle$ }{ $\langle submacro \rangle$ }`

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

`\lstKV@CSTwoArg{⟨value⟩}{⟨subdefinition⟩}`

⟨value⟩ 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.

```
\lst@Key{sensitive}\relax[t]{\lstKV@SetIf{#1}\lst@ifensitive}
\lst@AddToHookExe{SetLanguage}{\let\lst@ifensitive\iftrue}
```

The last line is equivalent to

```
\lst@AddToHook{SetLanguage}{\let\lst@ifensitive\iftrue}
\global\let\lst@ifensitive\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` in *TeX material*. Note that we could define and init the key as follows:

```
\lst@Key{sensitive}t[t]{\lstKV@SetIf{#1}\lst@ifensitive}
\lst@AddToHook{SetLanguage}{\let\lst@ifensitive\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*/*L^ATeX* 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

```
"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’ → ‘processing mode’. This escape to *L^ATeX* 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.

Table 3: Internal modes		
aspect	$\langle mode\ name \rangle$	Usage/We are processing ...
kernel	<code>\lst@nomode</code>	If this mode is active, T _E X’s ‘character table’ is present; the other implication is not true. Any other mode <i>may</i> imply that catcodes and/or definitions of characters are changed.
	<code>\lst@Pmode</code>	is a general processing mode. If active we are processing a listing, but haven’t entered a more special mode.
	<code>\lst@GPmode</code>	general purpose mode for language definitions.
pod	<code>\lst@PODmode</code>	... a POD—Perl specific.
escape	<code>\lst@TeXLmode</code>	... a comment line, but T _E X’s character table is present—except the EOL character, which is needed to terminate this mode.
	<code>\lst@TeXmode</code>	indicates that T _E X’s character table is present (except one user specified character, which is needed to terminate this mode).
directives	<code>\lst@CDmode</code>	indicates that the current line began with a compiler directive.
keywordcomments	<code>\lst@KCmode</code>	... a keyword comment.
	<code>\lst@KCSmode</code>	... a keyword comment which can be terminated by a semicolon only.
html	<code>\lst@insidemode</code>	Active if we are between < and >.
make	<code>\lst@makemode</code>	Used to indicate a keyword.

`\lst@NewMode` $\langle mode \text{ (control sequence)} \rangle$

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.

changed `\lst@EnterMode` $\langle mode \rangle\{\langle start tokens \rangle\}$

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 T_EX's character table is active.

Some variables show the internal state of processing. You are allowed to read them, but *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.

9.4 Hooks

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

deactivates/activates features for `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 **T_EX**'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 43 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 T_EX 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 <spec. token>
```

```
\lst@ProcessDigit <spec. token>
```

```
\lst@ProcessOther <spec. token>
```

```
\lst@ProcessTabulator
```

```
\lst@ProcessSpace
```

```
\lst@ProcessFormFeed
```

<spec. token> is supposed to do two things. Usually it expands to a printable version of the character. But if `\lst@UM` is equivalent to `\@empty`, *<spec. token>* 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{<character code>}{<parameter text>}{<definition>}
```

```
\lst@Let{<character code>}{<token>}
```

defines the specified character respectively assigns *<token>*. 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 T_EX’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.

data `\lst@lastother`

equals $\langle spec. token \rangle$ version of the last processed nonidentifier-character. Since programming languages redefine the standard character table, we use the original $\langle spec. token \rangle$. 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 $\langle spec. token \rangle$ 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 identifier 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 $\langle spec. token \rangle$ (obviously for character code 32) to test against `\lst@lastother`.

`\lstum@backslash`

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

`\lst@SaveOutputDef{ $\langle character code \rangle$ } $\langle macro \rangle$`

Stores the $\langle spec. token \rangle$ corresponding to $\langle character code \rangle$ in $\langle macro \rangle$. This is the only safe way to get a correct meaning to test against `\lst@lastother`, for example `\lst@SaveOutputDef{"5C}\lstum@backslash`.

You'll get a "runaway argument" error if $\langle character code \rangle$ 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.

global dimension `\lst@currlwidth`

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 `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`

<code>{\prefix}</code>	<code>k</code>
<code>{\name}</code>	<code>{keywords}</code>
<code>{\style name}</code>	<code>{keywordstyle}</code>
<code>{\style init}</code>	<code>\bfseries</code>
<code>{\default style name}</code>	<code>{keywordstyle}</code>
<code>{\working procedure}</code>	<code>{}</code>
<code>\l o</code>	<code>l</code>
<code>\d o</code>	<code>d</code>

installs either a keyword or ‘working’ class of identifiers according to whether `\working procedure` is empty.

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

If $\langle working procedure \rangle$ is not empty, this code is executed when reaching such an (user defined) identifier. $\langle working procedure \rangle$ 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 `\globally`. It's not sure whether working procedures are executed inside a (separate) group or not.

1 indicates a language key, i.e. the lists are reset every language selection. o stands for 'other' key. The keyword respectively working test is either installed at the `DetectKeyword` or `Output` hook according to $\langle d|o \rangle$.

```
\lst@InstallKeywords
    {\langle prefix \rangle}                                cs
    {\langle name \rangle}                                {texcs}
    {\langle style name \rangle}                          {texcsstyle}
    {\langle style init \rangle}                          \relax
    {\langle default style name \rangle}                  {keywordstyle}
    {\langle working procedure \rangle}                   see below
    \langle l|o \rangle                                   1
    \langle d|o \rangle                                   d
```

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

The working procedure of the example reads as follows.

```
{\ifx\lst@lastother\lstum@backslash
  \let\lst@thestyle\lst@texcsstyle
\fi}
```

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 `Comment`, `String`, and `Delim`. As you know, the latter and the first both provide the type 1, 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

```
\lst@AddTo\lst@stringtypes{,1}
```

Then we must provide the macro which takes the user supplied delimiter and makes appropriate definitions. The command name consists of the prefix `\lst@`, the delimiter name, DM for using dynamic modes, and `@` followed by the type.

```
\gdef\lst@StringDM@1#1#2\@empty#3#4#5{%
  \lst@CArg #2\relax\lst@DefDelimB{#1}{#3}{#4}{#5\lst@Lmodetrue}}
```

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

```

\lstset{string=[1]//}
\begin{lstlisting}
//_This_is_a_string.
This isn't a string.
This isn't a string.
\end{lstlisting>
```

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 `{\langle style \rangle \langle start tokens \rangle}`. 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
  {\langle 1st \rangle \langle 2nd \rangle \langle rest \rangle}           {//{}}
  \langle save 1st \rangle                                     \lst@c/0
  {\langle execute \rangle}                                   {}
  {\langle delim exe modetrue \rangle}                       {}
  {\langle delim exe modefalse \rangle}                     {}
  \langle start-delimiter macro \rangle                      #3
  \langle mode number \rangle                                {#1}
  {\langle style \rangle \langle start tokens \rangle}         {#5\lst@Lmodetrue}
```

defines `\langle 1st \rangle \langle 2nd \rangle \langle rest \rangle` as starting-delimiter. `\langle execute \rangle` is executed when the package comes to `\langle 1st \rangle`. `\langle delim exe modetrue \rangle` and `\langle delim exe modefalse \rangle` are executed only if the whole delimiter `\langle 1st \rangle \langle 2nd \rangle \langle rest \rangle` 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.

`\lst@DefDelimE`

```
{\1st\2nd}{\rest}}
\save 1st
\execute}
\delim exe modetrue}
\delim exe modefalse}
\end-delimiter macro}
\mode number}
```

Ditto for ending-delimiter with slight differences: `\delim exe modetrue` and `\delim exe modefalse` are executed depending on whether `\lst@mode` equals `\mode`.

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

`\lst@DefDelimBE`

followed by the same eight arguments as for `\lst@DefDelimB` and ...
`\end-delimiter macro`

This is a combination of `\lst@DefDelimB` and `\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

```
\gdef\lst@StringDM@d#1#2\@empty#3#4#5{%
  \lst@CArg #2\relax\lst@DefDelimBE}{-}{-}#3{#1}{#5}#4}
```

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

```
\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}}
```

Here we just need to use both `\lst@DefDelimB` and `\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.

```
\lst@Key{delim}\relax{\lst@DelimKey\@empty{#1}}
\lst@Key{moredelim}\relax{\lst@DelimKey\relax{#1}}
\lst@Key{deletedelim}\relax{\lst@DelimKey\@nil{#1}}
```

The service macro itself calls another macro with appropriate arguments.

```
\gdef\lst@DelimKey#1#2{%
  \lst@Delim{#2}\relax{Delim}\lst@delimtypes #1%
  {\lst@BeginDelim\lst@EndDelim}
  i\@empty{\lst@BeginIDelim\lst@EndIDelim}}
```

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.

changed \lst@Delim

```
<default style macro> \lst@commentstyle
[*][<type>][<style>][<type option>]]
<delimiter(s)>\relax #2\relax
{<delimiter name>} {Comment}
<delimiter types macro> \lst@commenttypes
\@empty|\@nil|\relax #1
{<begin- and end-delim macro>} {\lst@BeginComment\lst@EndComment}
<extra prefix> i
<extra conversion> \@empty
{<begin- and end-delim macro>} {\lst@BeginIComment\lst@EndIComment}
```

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@<name>[DM]<type>` 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 `<extra prefix>`, 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@<name>[DM]<type>`. If the user type starts with `<extra prefix>`, `<extra conversion>` might change the definition of `\lst@next` to choose a different conversion. The default is equivalent to `\lst@XConvert` with `\lst@false`.

Note that `<type>` never starts with `<extra prefix>` 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<whatever>
{<mode>} {<style>}{<start tokens>} <delimiter>\@empty
```



```
\lst@End<whatever>
    {\<mode>} \<delimiter>\@empty
```

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 4.16. 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 `\lsthk@PreSet` in any case.
4. Now you *might* want to (but need not) use `\lstset` to set some new values.
5. *<Do whatever you want.>*
6. Execute `\lst@Init\relax` 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\relax`
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.
10. Nothing.
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` $\langle macro \rangle \{ \langle T_{\text{E}}X \text{ material} \rangle \}$

adds $\langle T_{\text{E}}X \text{ material} \rangle$ globally to the contents of $\langle macro \rangle$.

`\lst@Extend` $\langle macro \rangle \{ \langle T_{\text{E}}X \text{ material} \rangle \}$

calls `\lst@AddTo` after the first token of $\langle T_{\text{E}}X \text{ material} \rangle$ is `\expandedafter`. For example, `\lst@Extend \a \b` merges the contents of the two macros and stores it globally in `\a`.

`\lst@lAddTo` $\langle macro \rangle \{ \langle T_{\text{E}}X \text{ material} \rangle \}$

`\lst@lExtend` $\langle macro \rangle \{ \langle T_{\text{E}}X \text{ material} \rangle \}$

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

`\lst@DeleteKeysIn` $\langle macro \rangle \langle macro \text{ (keys to remove)} \rangle$

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` $\langle macro \rangle \langle macro \text{ (containing replacement list)} \rangle$

`\lst@ReplaceInArg` $\langle macro \rangle \{ \langle replacement \text{ list} \rangle \}$

The replacement list has the form $a_1 b_1 \dots a_n b_n$, 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 `\@empty`. Each sequence a_i inside the first macro is (locally) replaced by b_i . The suffix `Arg` refers to the *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` $\{ \langle character \text{ sequence} \rangle \} \langle macro \rangle \{ \langle then \rangle \} \{ \langle else \rangle \}$

$\langle then \rangle$ is executed if $\langle character \text{ sequence} \rangle$ is a substring of the contents of $\langle macro \rangle$. Otherwise $\langle else \rangle$ is called.

`\lst@ifoneof` $\langle character \text{ sequence} \rangle \backslash relax \langle macro \rangle \{ \langle then \rangle \} \{ \langle else \rangle \}$

`\relax` terminates the first parameter here since it is faster than enclosing it in braces. $\langle macro \rangle$ contains a comma separated list of identifiers. If the character sequence is one of these identifiers, $\langle then \rangle$ is executed, and otherwise $\langle else \rangle$.

`\lst@Swap{<tok1>}{<tok2>}`

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

`\lst@ifnextchars<macro>{<then>}{<else>}`

`\lst@ifnextcharsarg{<character sequence>}{<then>}{<else>}`

Both macros execute either `<then>` or `<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 L^AT_EX's `\@ifnextchar`: We remove the characters behind the arguments until it is possible to decide which part must be executed. However, we save these characters in the macro `\lst@eaten`, so they can be inserted using `<then>` or `<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 `<macro>`, but all characters become active. The string *must not* contain a begin group, end group or escape character (`{}` or `\`); it may contain a left brace, right brace or backslash with other meaning (= catcode). This command would be quite surplus if `<character sequence>` is not already read by T_EX since such catcodes can be changed easily. It is explicitly allowed that the characters have been read, e.g. in `\def\test{\lst@DefActive\temp{ABC}}!`

Note that this macro changes `\lccodes 0–9` without restoring them.

`\lst@DefOther<macro>{<character sequence>}`

stores `<character sequence>` in `<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, `\{ Chip \}` leads to `{Chip}` where all catcodes are 12—internally the primitive `\meaning` is used.

11.2 Character tables manipulated

`\lst@SaveDef{<character code>}<macro>`

Saves the current definition of the specified character in `<macro>`. You should always save a character definition before you redefine it! And use the saved version instead of writing directly `\lst@Process...`—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 *where* to alter the character table. Hook material at `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.

SelectCharTable= $\langle T_{\text{E}}X$ code \rangle

MoreSelectCharTable= $\langle T_{\text{E}}X$ code \rangle

uses $\langle T_{\text{E}}X$ code \rangle (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 .log file if language `useless` is active:

```
\lstdefinlanguage{useless}
  {SelectCharTable=\lst@DefSaveDef{46}% save chr(46) ...
    \lst@point % ... in \lst@point and ...
    {\message{.}\lst@point}% ... use new definition
  }
```

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}}
\lstdefinlanguage[2]{useless}
  {SelectCharTable=\lst@DefSaveDef{46}\lst@point
    {\global\advance\lst@points\@ne \lst@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.

$\backslash\text{lst@CArg}\langle\text{active characters}\rangle\backslash\text{relax}\langle\text{macro}\rangle$

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

$\backslash\text{lst@CArgX}\langle\text{characters}\rangle\backslash\text{relax}\langle\text{macro}\rangle$

makes $\langle characters \rangle$ active before calling $\backslash\text{lst@CArg}$.

$\backslash\text{lst@CDef}\{\langle 1st \rangle\langle 2nd \rangle\{\langle rest \rangle\}\}\langle save 1st \rangle\{\langle execute \rangle\}\{\langle pre \rangle\}\{\langle post \rangle\}$

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

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

$\backslash\text{lst@CDefX}\langle 1st \rangle\langle 2nd \rangle\{\langle rest \rangle\}\langle save 1st \rangle\{\langle execute \rangle\}\{\langle pre \rangle\}\{\langle post \rangle\}$

Ditto except that we execute $\langle pre \rangle$ and $\langle post \rangle$ without the original string if we reach $\langle 1st \rangle\langle 2nd \rangle\langle rest \rangle$. 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{#1}#2%
    {#3}%
    {\let\lst@bnext\lst@CArgEmpty
     \lst@ifmode #4\else
       #5%
       \def\lst@bnext{#6{#7}{#8}}%
     \fi
     \lst@bnext}%
  \@empty}
```

You got it?

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 `\@gtempa` 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,
- `\lsts@` for saved character meanings,
- `\lsthk@<name of hook>` holds hook material,
- `\lst<prefix>@` for various kinds of keywords and working identifiers.
- `\lstlang@<language>@<dialect>` contains a language and
- `\lststy@<the style>` contains style definition,
- `\lstpatch@<aspect>` to patch an aspect,
- `\lsta@<language>$<dialect>` contains alias,
- `\lsta@<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. `\lst@AddTo`. A macro with suffix `@` is the main working-procedure for another definition, for example `\lstinputlisting@` does the main work for `\lstinputlisting`.

Preamble All files generated from this `listings.dtx` will get a header.

```
1 %% Please read the software license in listings.dtx or listings.pdf.
2 %%
3 %% (w)(c) 1996 -- 2004 Carsten Heinz and/or any other author
4 %% listed elsewhere in this file.
5 %%
6 %% This file is distributed under the terms of the LaTeX Project Public
7 %% License from CTAN archives in directory macros/latex/base/lppl.txt.
8 %% Either version 1.0 or, at your option, any later version.
9 %%
10 %% Permission is granted to modify this file. If your changes are of
11 %% general interest, please contact the address below.
12 %%
13 %% Send comments and ideas on the package, error reports and additional
14 %% programming languages to <cheinz@gmx.de>.
15 %%
```

Identification All files will have same date and version.

```
16 \def\filedate{2004/02/13}
17 \def\fileversion{1.2}
```

What we need and who we are.

```
18 <*kernel>
19 \NeedsTeXFormat{LaTeX2e}
20 \ProvidesPackage{listings}
21      [\filedate\space\fileversion\space(Carsten Heinz)]
```

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

```
22 \def\lst@CheckVersion#1{\edef\reserved@a{#1}%
23   \ifx\lst@version\reserved@a \expandafter\@gobble
24   \else \expandafter\@firstofone \fi}
25 \let\lst@version\fileversion
26 </kernel>
```

For example by the miscellaneous file

```
27 <*misc>
28 \ProvidesFile{lstmisc.sty}
29      [\filedate\space\fileversion\space(Carsten Heinz)]
30 \lst@CheckVersion\fileversion
31   {\typeout{^^J%
32     ***^^J%
33     *** This file requires 'listings.sty' version \fileversion.^^J%
34     *** You have a serious problem, so I'm exiting ...^^J%
35     ***^^J}%
36   \batchmode \@@end}
37 </misc>
```

or by the dummy patch.

```
38 <*patch>
39 \ProvidesFile{lstpatch.sty}
40      [\filedate\space\fileversion\space(Carsten Heinz)]
41 \lst@CheckVersion\lst@version{}
42 </patch>
```

```

43 <*doc>
44 \ProvidesPackage{lstdoc}
45           [\filedate\space\fileversion\space(Carsten Heinz)]
46 </doc>

```

Category codes We define two macros to ensure correct catcodes when we input other files of the listings package.

`\lst@InputCatcodes` @ and " become letters. Tabulators and EOLs are ignored; this avoids unwanted spaces—in the case I’ve forgotten a comment character.

```

47 <*kernel>
48 \def\lst@InputCatcodes{%
49     \makeatletter \catcode'\ "12%
50     \catcode'\ ^^@ \active
51     \catcode'\ ^^I 9%
52     \catcode'\ ^^L 9%
53     \catcode'\ ^^M 9%
54     \catcode'\ %14%
55     \catcode'\ ^^ \active}

```

`\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 `typehtml` package, which is resolved by `\lccode'\ /' \ /` below.

```

56 \def\lst@RestoreCatcodes#1{%
57     \ifx\relax#1\else
58         \noexpand\catcode'\noexpand#1\the\catcode'#1\relax
59         \expandafter\lst@RestoreCatcodes
60     \fi}
61 \edef\lst@RestoreCatcodes{%
62     \noexpand\lccode'\noexpand\ /' \noexpand\ /%
63     \lst@RestoreCatcodes"\ ^^I \ ^^M \ ^^@ \relax}

```

Now we are ready for

```

64 \lst@InputCatcodes
65 \AtEndOfPackage{\lst@RestoreCatcodes}
66 </kernel>

```

Statistics

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

```

\lst@ReportAllocs 67 <*info>
68 \def\lst@GetAllocs{%
69     \edef\lst@allocs{%
70         0\noexpand\count\the\count10,1\noexpand\dimen\the\count11,%
71         2\noexpand\skip\the\count12,3\noexpand\muskip\the\count13,%
72         4\noexpand\box\the\count14,5\noexpand\toks\the\count15,%
73         6\noexpand\read\the\count16,7\noexpand\write\the\count17}}
74 \def\lst@ReportAllocs{%
75     \message{^^JAllocs:}\def\lst@temp{none}%
76     \expandafter\lst@ReportAllocs@\lst@allocs,\z@\relax\z@,}
77 \def\lst@ReportAllocs@#1#2#3,{%
78     \ifx#2\relax \message{\lst@temp^^J}\else
79         \@tempcnta\count1#1\relax \advance\@tempcnta -#3\relax
80         \ifnum\@tempcnta=\z@\else

```

```

81         \let\lst@temp\@empty
82         \message{\the\@tempcnta \string#2,}%
83     \fi
84     \expandafter\lst@ReportAllocs@
85 \fi}
86 \lst@GetAllocs

87 </info>

```

Miscellaneous

```

\@lst Just a definition to save memory space.
88 <*kernel>
89 \def\@lst{lst}
90 </kernel>

```

13 General problems

All definitions in this section belong to the kernel.

```
91 <*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 `def` we could say

```

\def \lst@temp#1def#2\relax{%
  \ifx \@empty#2\@empty
    % "def" is not a substring
  \else
    % "def" is a substring
  \fi}

```

```
\lst@temp <another string>def\relax
```

When `TEX` passes the arguments `#1` and `#2`, the second is empty if and only if `def` is not a substring. Without the additional `def\relax`, one would get a “runaway argument” error if `<another string>` doesn't contain `def`.

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

```

\def \lst@temp#1,key,#2\relax{%
  \ifx \@empty#2\@empty
    % 'key' is not a keyword
  \else
    % 'key' is a keyword
  \fi}

```

```
\lst@temp,<list of keywords>,key,\relax
```

This works very well and is quite fast. But we can reduce run time in the case that `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:

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

```
\lst@temp,<list of keywords>,key,\@empty\relax
```

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

`\lst@ifsubstring` The implementation should be clear from the discussion above.

```
92 \def\lst@ifsubstring#1#2{%
93   \def\lst@temp##1#1##2##3\relax{%
94     \ifx \@empty##2\expandafter\@secondoftwo
95       \else \expandafter\@firstoftwo \fi}%
96   \expandafter\lst@temp#2#1\@empty\relax}
```

`\lst@ifoneof` Ditto.

```
97 \def\lst@ifoneof#1\relax#2{%
98   \def\lst@temp##1,#1,##2##3\relax{%
99     \ifx \@empty##2\expandafter\@secondoftwo
100       \else \expandafter\@firstoftwo \fi}%
101   \expandafter\lst@temp\expandafter,#2,#1,\@empty\relax}
```

Removed: One day, if there is need for a case insensitive key(word) test again, we can use two `\uppercase` to normalize the first parameter:

```
\def\lst@ifoneofinsensitive#1\relax#2{%
  \uppercase{\def\lst@temp##1,#1,##2##3\relax{%
    \ifx \@empty##2\expandafter\@secondoftwo
      \else \expandafter\@firstoftwo \fi}%
  \uppercase{%
    \expandafter\lst@temp\expandafter,#2,#1,\@empty\relax}
```

Here we assume that macro #2 already contains capital characters only, see the definition of `\lst@makeMacroUppercase` at the very end of section 16.1. If we *must not* assume that, we could simply insert an `\expandafter` between the second `\uppercase` and the following brace. But this slows down the tests!

`\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`.

```
102 \def\lst@deletekeysin#1#2{%
103   \expandafter\lst@deletekeysin@\expandafter#1#2,\relax,}
‘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.
104 \def\lst@deletekeysin@#1#2,{%
105   \ifx\relax#2\@empty
106     \expandafter\@firstoftwo\expandafter\lst@removecommas
107   \else
108     \ifx\@empty#2\@empty\else
```

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.

```

109         \def\lst@temp##1,#2,##2{%
110             ##1%
111             \ifx\@empty##2\@empty\else
112                 \expandafter\lst@temp\expandafter,%
113             \fi ##2}%
114         \edef#1{\expandafter\lst@temp\expandafter,#1,#2,\@empty}%
115     \fi
116 \fi
117 \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.

```

\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

```

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

```

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

```

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

```

\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}}

```

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

```

\lst@ReplaceInArg 120 \def\lst@ReplaceIn#1#2{%
121     \expandafter\lst@ReplaceIn@\expandafter#1#2\@empty\@empty}
122 \def\lst@ReplaceInArg#1#2{\lst@ReplaceIn@#1#2\@empty\@empty}

```

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

```

123 \def\lst@ReplaceIn@#1#2#3{%
124     \ifx\@empty#3\relax\else
125         \def\lst@temp##1#2##2{%
126             \ifx\@empty##2%
127                 \lst@lAddTo#1{##1}%
128             \else
129                 \lst@lAddTo#1{##1#3}\expandafter\lst@temp
130             \fi ##2}%
131         \let\@tempa#1\let#1\@empty
132         \expandafter\lst@temp\@tempa#2\@empty
133         \expandafter\lst@ReplaceIn@\expandafter#1%
134     \fi}

```

13.2 Flow of control

```

\@gobblethree is defined if and only if undefined.
135 \providecommand*\@gobblethree[3]{}

\lst@GobbleNil
136 \def\lst@GobbleNil#1\@nil{}

\lst@Swap is just this:
137 \def\lst@Swap#1#2{#2#1}

\lst@if A general \if for temporary use.
\lst@true 138 \def\lst@true{\let\lst@if\iftrue}
\lst@false 139 \def\lst@false{\let\lst@if\iffalse}
140 \lst@false

\lst@ifnextcharsarg is quite easy: We define a macro and call \lst@ifnextchars.
141 \def\lst@ifnextcharsarg#1{%
142   \def\lst@tofind{#1}\lst@ifnextchars\lst@tofind}

\lst@ifnextchars We save the arguments and start a loop.
143 \def\lst@ifnextchars#1#2#3{%
144   \let\lst@tofind#1\def\@tempa{#2}\def\@tempb{#3}%
145   \let\lst@eaten\@empty \lst@ifnextchars@

   Expand the characters we are looking for.
146 \def\lst@ifnextchars@{\expandafter\lst@ifnextchars@@\lst@tofind\relax}

   Now we can refine \lst@tofind and append the input character #3 to \lst@eaten.
147 \def\lst@ifnextchars@@#1#2\relax#3{%
148   \def\lst@tofind{#2}\lst@lAddTo\lst@eaten{#3}%
149   \ifx#1#3%

   If characters are the same, we either call \@tempa or continue the test.
150     \ifx\lst@tofind\@empty
151       \let\lst@next\@tempa
152     \else
153       \let\lst@next\lst@ifnextchars@
154     \fi
155     \expandafter\lst@next
156   \else

   If the characters are different, we call \@tempb.
157     \expandafter\@tempb
158   \fi}

\lst@ifnextcharactive We compare the character #3 with its active version \lowercase{~}. Note that
the right brace between \ifx~ and #3 ends the \lowercase. The \endgroup
restores the \lccode.
159 \def\lst@ifnextcharactive#1#2#3{%
160   \begingroup \lccode'\~='#3\lowercase{\endgroup
161   \ifx~}#3%
162     \def\lst@next{#1}%
163   \else
164     \def\lst@next{#2}%
165   \fi \lst@next #3}

```

`\lst@for` A for-loop with expansion of the loop-variable.

```

166 \def\lst@for#1\do#2{%
167   \def\lst@forbody##1{#2}%
168   \@for\lst@forvar:=#1\do
169   {\expandafter\lst@forbody\expandafter{\lst@forvar}}}
```

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

```
\def\lst@MakeActive#1{\lccode'\~='#1\lowercase{\def\lst@arg{~}}}
```

The `\lowercase` changes the ASCII code of `~` to the one of `#1` since we have said that `~` is the lower case version of `#1`. 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.

```

170 \def\lst@MakeActive#1{%
171   \let\lst@temp\@empty \lst@MakeActive@#1%
172   \relax\relax\relax\relax\relax\relax\relax\relax\relax}
```

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`.

```

173 \begingroup
174 \catcode'\^^@=\active \catcode'\^^A=\active \catcode'\^^B=\active
175 \catcode'\^^C=\active \catcode'\^^D=\active \catcode'\^^E=\active
176 \catcode'\^^F=\active \catcode'\^^G=\active \catcode'\^^H=\active
```

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.

```

177 \gdef\lst@MakeActive@#1#2#3#4#5#6#7#8#9{\let\lst@next\relax
178   \ifx#1\relax
179     \else \lccode'\^^@='#1%
```

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.

```

180   \ifx#2\relax
181     \lowercase{\lst@lAddTo\lst@temp{^^@}}%
```

```

182 \else \lccode'\^^A='#2%
183 \ifx#3\relax
184 \lowercase{\lst@lAddTo\lst@temp{^^@^^A}}%
185 \else \lccode'\^^B='#3%
186 \ifx#4\relax
187 \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B}}%
188 \else \lccode'\^^C='#4%
189 \ifx#5\relax
190 \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C}}%
191 \else \lccode'\^^D='#5%
192 \ifx#6\relax
193 \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C^^D}}%
194 \else \lccode'\^^E='#6%
195 \ifx#7\relax
196 \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C^^D^^E}}%
197 \else \lccode'\^^F='#7%
198 \ifx#8\relax
199 \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C^^D^^E^^F}}%
200 \else \lccode'\^^G='#8%
201 \ifx#9\relax
202 \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C^^D^^E^^F^^G}}%

```

If nine characters are present, we append (lower case versions of) nine active characters and call this macro again via redefining `\lst@next`.

```

203 \else \lccode'\^^H='#9%
204 \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C^^D^^E^^F^^G^^H}}%
205 \let\lst@next\lst@MakeActive@
206 \fi \fi \fi \fi \fi \fi \fi \fi \fi
207 \lst@next}
208 \endgroup

```

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`.

```

209 \def\lst@DefActive#1#2{\lst@MakeActive{#2}\let#1\lst@temp}

```

`\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 Niepraschk—not in this context, but that doesn’t matter.

```

210 \def\lst@DefOther#1#2{%
211 \begingroup \def#1{#2}\escapechar\m@ne \expandafter\endgroup
212 \expandafter\lst@DefOther@\meaning#1\relax#1}
213 \def\lst@DefOther@#1>#2\relax#3{\edef#3{\zap@space#2 \@empty}}

```

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 `fancyvrb` mode.

`\lst@InsideConvert{<TEX material (already read)>}`

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{<TEX material (already read)>}`

appends a ‘verbatim’ version of the argument to `\lst@arg`. Here `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` We call a submacro (similar to `\zap@space`) to preserve single spaces which are replaced by active spaces.

```
214 \def\lst@InsideConvert#1{\lst@InsideConvert@#1 \empty}
215 \begingroup \lccode'\~=' \relax \lowercase{%
```

We make `#1` active and append these characters (plus an active space) to `\lst@arg`. If we haven’t found the end `\empty` of the input, we continue the process.

```
216 \gdef\lst@InsideConvert@#1 #2{%
217   \lst@MakeActive{#1}%
218   \ifx\empty#2%
219     \lst@lExtend\lst@arg{\lst@temp}%
220   \else
221     \lst@lExtend\lst@arg{\lst@temp~}%
222     \expandafter\lst@InsideConvert@
223   \fi #2}
```

Finally we end the `\lowercase` and close a group.

```
224 }\endgroup
```

`\lst@XConvert` Check for an argument ...

```
225 \def\lst@XConvert{\@ifnextchar\bgroup \lst@XConvertArg\lst@XConvert@}
... , convert the argument, add it together with group delimiters to \lst@arg, and
we continue the conversion.
```

```
226 \def\lst@XConvertArg#1{%
227   {\lst@false \let\lst@arg\empty
228    \lst@XConvert#1\@nil
229    \global\let\@gtempa\lst@arg}%
230   \lst@lExtend\lst@arg{\expandafter{\@gtempa}}%
231   \lst@XConvertNext}
```

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.

```

232 \def\lst@XConvert@#1{%
233   \ifx\@nil#1\else
234     \begingroup\lccode'\~='#1\lowercase{\endgroup
235     \lst@lAddTo\lst@arg~}%
236     \expandafter\lst@XConvertNext
237   \fi}
238 \def\lst@XConvertNext{%
239   \lst@if \expandafter\lst@XConvertX
240   \else \expandafter\lst@XConvert \fi}

```

Now we make only the first character active.

```

241 \def\lst@XConvertX#1{%
242   \ifx\@nil#1\else
243     \lst@XConvertX@#1\relax
244     \expandafter\lst@XConvert
245   \fi}
246 \def\lst@XConvertX@#1#2\relax{%
247   \begingroup\lccode'\~='#1\lowercase{\endgroup
248   \lst@XCConvertX@@~}{#2}}
249 \def\lst@XCConvertX@@#1#2{\lst@lAddTo\lst@arg{{#1#2}}}

```

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>}<alias><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.

`\lst@Require` Initialize variables (if required items aren't empty), ...

```

250 \def\lst@Require#1#2#3#4#5{%
251     \begingroup
252     \aftergroup\lst@true
253     \ifx\@empty#3\@empty\else
254         \def\lst@prefix{#2}\let\lst@require\@empty
... and for each nonempty item: determine alias and add it to \lst@require if
it isn't loaded.
255         \edef\lst@temp{\expandafter\zap@space#3 \@empty}%
256         \lst@for\lst@temp\do{%
257             \ifx\@empty##1\@empty\else \lstKV@OptArg[] {##1}{%
258                 #4[####1]{####2}%
259                 \@ifundefined{\@lst\lst@prefix @\lst@malias $\lst@oalias}%
260                 {\edef\lst@require{\lst@require,\lst@malias $\lst@oalias}}%
261                 {}}%
262             \fi}%
Init things and input files if and as long as it is necessary.
263     \global\let\lst@loadaspects\@empty
264     \lst@InputCatcodes
265     \ifx\lst@require\@empty\else
266         \lst@for{#5}\do{%
267             \ifx\lst@require\@empty\else
268                 \InputIfFileExists{##1}{\fi}%
269             \fi}%
270     \fi
Issue error and call \lst@false (after closing the local group) if some items weren't
loadable.
271     \ifx\lst@require\@empty\else
272         \PackageError{Listings}{Couldn't load requested #1}%
273         {The following #1s weren't loadable:^^J\@spaces
274         \lst@require^^JThis may cause errors in the sequel.}%
275     \aftergroup\lst@false
276     \fi
Request aspects.
277     \ifx\lst@loadaspects\@empty\else
278         \lst@RequireAspects\lst@loadaspects
279     \fi
280     \fi
281     \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>$<sub>`.

```

282 \def\lst@ifrequired[#1]#2{%
283     \lst@NormedDef\lst@temp{[#1]#2}%
284     \expandafter\lst@ifrequired@\lst@temp\relax}
285 \def\lst@ifrequired@[#1]#2\relax#3{%
286     \lst@ifoneof #2$#1\relax\lst@require
287     {\lst@DeleteKeysIn@\lst@require#2$#1,\relax,%
288     \global\expandafter\let
289     \csname@\lst\lst@prefix @#2$#1\endcsname\@empty
290     #3}}

```



```

\lst@require
291 \let\lst@require\@empty

\lst@NoAlias just defines \lst@oalias and \lst@malias.
292 \def\lst@NoAlias[#1]#2{%
293     \lst@NormedDef\lst@oalias{#1}\lst@NormedDef\lst@malias{#2}}

\lst@LAS
294 \gdef\lst@LAS#1#2#3#4#5#6#7{%
295     \lst@Require{#1}{#2}{#3}{#4}{#5}
296     #4#3%
297     \@ifundefined{lst#2@\lst@malias$\lst@oalias}%
298     {\PackageError{Listings}%
299     {#1 \ifx\@empty\lst@oalias\else \lst@oalias\space of \fi
300     \lst@malias\space undefined}%
301     {The #1 is not loadable. \@ehc}}%
302     {#6\csname\@lst#2@\lst@malias $\lst@oalias\endcsname #7}}

\lst@RequireAspects make use of the just developped definitions.
\lstloadaspects 303 \def\lst@RequireAspects#1{%
304     \lst@Require{aspect}{asp}{#1}\lst@NoAlias\lstaspectfiles}
305 \let\lstloadaspects\lst@RequireAspects

\lstaspectfiles This macro is defined if and only if it's undefined yet.
306 \@ifundefined{lstaspectfiles}
307     {\newcommand\lstaspectfiles{lstmisc0.sty,lstmisc.sty}}{}

\lst@DefDriver Test the next character and reinsert the arguments.
308 \gdef\lst@DefDriver#1#2#3#4{%
309     \@ifnextchar[{\lst@DefDriver@{#1}{#2}{#3#4}}%
310     {\lst@DefDriver@{#1}{#2}{#3#4[]}}
311     We set \lst@if locally true if the item has been requested.
312     \gdef\lst@DefDriver@#1#2#3#4[#5]#6{%
313         \def\lst@name{#1}\let\lst@if#4%
314         \lst@NormedDef\lst@driver{\@lst#2@#6$#5}%
315         \lst@ifrequired[#5]{#6}{\begingroup \lst@true}%
316         {\begingroup}%
317         \lst@setcatcodes
318         \@ifnextchar[{\lst@XDefDriver{#1}{#3}{\lst@DefDriver@@#3}}
319         \lst@if
320         \global\@namedef{\lst@driver}{#1{#2}}%
321         \fi
322         \endgroup
323         \@ifnextchar[\lst@XXDefDriver\@empty}

```

Note that \lst@XDefDriver takes optional ‘base’ arguments, but eventually calls \lst@DefDriver@@. We define the item (in case of need), and \endgroup resets some catcodes and \lst@if, i.e. \lst@XXDefDriver knows whether called by a public or internal command.

We get the aspect argument, and (if not empty) load the aspects immediately if called by a public command or extend the list of required aspects or simply ignore the argument if the item leaves undefined.

```

324 \gdef\lst@XXDefDriver[#1]{%
325     \ifx\@empty#1\@empty\else
326         \lst@if
327             \lstloadaspects{#1}%
328         \else
329             \@ifundefined{\lst@driver}{}%
330             {\xdef\lst@loadaspects{\lst@loadaspects,#1}}%
331         \fi
332     \fi}

```

We insert an additional ‘also’key=value pair.

```

333 \gdef\lst@XDefDriver#1#2[#3]#4#5{\lst@DefDriver@@#2{also#1=[#3]#4,#5}}

```

13.6 Aspect commands

This section contains commands used in defining ‘lst-aspects’.

`\lst@UserCommand` is mainly equivalent to `\gdef`.

```

334 \!info\let\lst@UserCommand\gdef
335 \!info\def\lst@UserCommand#1{\message{\string#1,}\gdef#1}

```

`\lst@BeginAspect` A straight-forward implementation:

```

336 \newcommand*\lst@BeginAspect[2][{}]{%
337     \def\lst@curraspect{#2}%
338     \ifx \lst@curraspect\@empty
339         \expandafter\lst@GobbleAspect
340     \else

```

If *aspect name* is not empty, there are certain other conditions not to define the aspect (as described in section 9.2).

```

341 \!info\let\lst@next\@empty
342 \!info\def\lst@next{%
343 \!info\message{^^JDefine lst-aspect ‘#2’:}\lst@GetAllocs}%
344 \lst@ifRequired[] {#2}%
345 {\lst@RequireAspects{#1}%
346 \lst@if\else \let\lst@next\lst@GobbleAspect \fi}%
347 {\let\lst@next\lst@GobbleAspect}%
348 \expandafter\lst@next
349 \fi}

```

`\lst@EndAspect` finishes an aspect definition.

```

350 \def\lst@EndAspect{%
351     \csname\@lst_patch@\lst@curraspect\endcsname
352 \!info\lst@ReportAllocs
353     \let\lst@curraspect\@empty}

```

`\lst@GobbleAspect` drops all code up to the next `\lst@EndAspect`.

```

354 \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.

```

355 \def\lst@Key#1#2{%
356   \info      \message{#1,}%
357   \@ifnextchar[{\lstKV@def{#1}{#2}}%
358             {\def\lst@temp{\lst@Key@{#1}{#2}}
359             \afterassignment\lst@temp
360             \global\@namedef{KV@\@lst @#1}###1}}

```

Now comes a renamed and modified copy from a keyval macro: We need global key definitions.

```

361 \def\lstKV@def#1#2[#3]{%
362   \global\@namedef{KV@\@lst @#1@default\expandafter}\expandafter
363   {\csname KV@\@lst @#1\endcsname{#3}}}%
364   \def\lst@temp{\lst@Key@{#1}{#2}}\afterassignment\lst@temp
365   \global\@namedef{KV@\@lst @#1}##1}

```

We initialize the key if the first token of #2 is not `\relax`.

```

366 \def\lst@Key@#1#2{%
367   \ifx\relax#2\@empty\else
368     \begingroup \globaldefs\@ne
369     \csname KV@\@lst @#1\endcsname{#2}%
370   \endgroup
371   \fi}

```

`\lst@UseHook` is very, very, ..., very (hundreds of times) easy.

```

372 \def\lst@UseHook#1{\csname\@lst hk@#1\endcsname}

```

`\lst@AddToHook` All use the same submacro.

```

\lst@AddToHookExe 373 \def\lst@AddToHook{\lst@ATH@\iffalse\lst@AddTo}
\lst@AddToHookAtTop 374 \def\lst@AddToHookExe{\lst@ATH@\iftrue\lst@AddTo}
375 \def\lst@AddToHookAtTop{\lst@ATH@\iffalse\lst@AddToAtTop}

```

If and only if the boolean value is true, the hook material is executed globally.

```

376 \long\def\lst@ATH@#1#2#3#4{%
377   \ifundefined{\@lst hk@#3}{%
378   \info      \message{^^Jnew hook ' #3', ^^J}%
379   \expandafter\gdef\csname\@lst hk@#3\endcsname{}}{}}%
380   \expandafter#2\csname\@lst hk@#3\endcsname{#4}%
381   \def\lst@temp{#4}%
382   #1% \iftrue|false
383   \begingroup \globaldefs\@ne \lst@temp \endgroup
384   \fi}

```

`\lst@AddTo` Note that the definition is global!

```

385 \long\def\lst@AddTo#1#2{%
386   \expandafter\gdef\expandafter#1\expandafter{#1#2}}

```

`\lst@AddToAtTop` We need a couple of `\expandafters` now. Simply note that we have

`\expandafter\gdef\expandafter#1\expandafter{\lst@temp (contents of #1)}` after the ‘first phase’ of expansion.

```

387 \def\lst@AddToAtTop#1#2{\def\lst@temp{#2}%
388   \expandafter\expandafter\expandafter\gdef
389   \expandafter\expandafter\expandafter#1%
390   \expandafter\expandafter\expandafter{\expandafter\lst@temp#1}}

```

```

\lst@lAddTo A local version of \lst@AddTo ...
391 \def\lst@lAddTo#1#2{\expandafter\def\expandafter#1\expandafter{#1#2}}

\lst@Extend ... and here we expand the first token of the second argument first.
\lst@lExtend 392 \def\lst@Extend#1#2{%
393     \expandafter\lst@AddTo\expandafter#1\expandafter{#2}}
394 \def\lst@lExtend#1#2{%
395     \expandafter\lst@lAddTo\expandafter#1\expandafter{#2}}

```

To do: This should never be changed to

```

\def\lst@Extend#1{%
    \expandafter\lst@AddTo\expandafter#1\expandafter}
\def\lst@lExtend#1{%
    \expandafter\lst@lAddTo\expandafter#1}

```

The first is not equivalent in case that the second argument is a single (= non-braced) control sequence, and the second isn't in case of a braced second argument.

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.

```

396 \RequirePackage{keyval}[1997/11/10]

```

```

\lstKV@TwoArg Define temporary macros and call with given arguments #1. We add empty argu-
\lstKV@ThreeArg ments for the case that the user doesn't provide enough.
\lstKV@FourArg 397 \def\lstKV@TwoArg#1#2{\gdef\@gtempa##1##2{#2}\@gtempa#1{}}
398 \def\lstKV@ThreeArg#1#2{\gdef\@gtempa##1##2##3{#2}\@gtempa#1{}}
399 \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.
400 \def\lstKV@OptArg[#1]#2#3{%
401     \gdef\@gtempa[##1]##2{#3}\lstKV@OptArg@{#1}#2\@}
402 \def\lstKV@OptArg@#1{\@ifnextchar[\lstKV@OptArg@@{\lstKV@OptArg@@[#1]}}
403 \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.
404 \def\lstKV@XOptArg[#1]#2#3{%
405     \global\let\@gtempa#3\lstKV@OptArg@{#1}#2\@}

```

```

\lstKV@CSTwoArg Just define temporary macro and call it.
406 \def\lstKV@CSTwoArg#1#2{%
407     \gdef\@gtempa##1,##2,##3\relax{#2}%
408     \@gtempa#1,,\relax}

```

`\lstKV@SetIf` We simply test the lower case first character of #1.

```
409 \def\lstKV@SetIf#1{\lstKV@SetIf@#1\relax}
410 \def\lstKV@SetIf@#1#2\relax#3{\lowercase{%
411     \expandafter\let\expandafter#3%
412     \csname if\ifx #1t\true\else false\fi\endcsname}
```

`\lstKV@SwitchCases` is implemented as a substring test.

```
413 \def\lstKV@SwitchCases#1#2#3{%
414     \def\lst@temp##1\|#1##2\|##3##4\@nil{%
415         \ifx\@empty##3%
416             #3%
417         \else
418             ##2%
419         \fi
420     }%
421     \lst@temp\|#2\|#1&\|\@empty\@nil}
```

`\lstset` Finally this main user interface macro. We change catcodes for reading the argument.

```
422 \lst@UserCommand\lstset{\begingroup \lst@setcatcodes \lstset@}
423 \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.

```
424 \def\lst@setcatcodes{\makeatletter \catcode'\ "=12\relax
425                               \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.

```
426 \def\lst@NewMode#1{%
427     \ifx\@undefined#1%
428         \lst@mode\lst@newmode\relax \advance\lst@mode\@ne
429         \xdef\lst@newmode{\the\lst@mode}%
430         \global\chardef#1=\lst@mode
431         \lst@mode\lst@nomode
432     \fi}
```

`\lst@mode` We allocate the counter and the first mode.

```
\lst@nomode 433 \newcount\lst@mode
434 \def\lst@newmode{\m@ne}% init
435 \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, ...

```
436 \def\lst@UseDynamicMode{%
```

```

437 \tempcnta\lst@dynamicmode\relax \advance\tempcnta\@ne
438 \edef\lst@dynamicmode{\the\tempcnta}%
439 \expandafter\lst@Swap\expandafter{\expandafter{\lst@dynamicmode}}}
... initialized each listing with the current 'value' of \lst@newmode.
440 \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.

```

441 \def\lst@EnterMode#1#2{%
442   \bgroup \lst@mode=#1\relax #2%
443   \lst@FontAdjust
444   \lst@lAddTo\lst@entermodes{\lst@EnterMode{#1}{#2}}
445 \lst@AddToHook{InitVars}{\let\lst@entermodes\@empty}
446 \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.

```

447 \def\lst@LeaveMode{%
448   \ifnum\lst@mode=\lst@nomode\else
449     \egroup \expandafter\lsthk@EndGroup
450   \fi}
451 \lst@AddToHook{EndGroup}{}% init

```

\lst@InterruptModes We put the current mode sequence on a stack and leave all modes.

```

452 \def\lst@InterruptModes{%
453   \lst@Extend\lst@modestack{\expandafter{\lst@entermodes}}%
454   \lst@LeaveAllModes}
455 \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).

```

456 \def\lst@ReenterModes{%
457   \ifx\lst@modestack\@empty\else
458     \lst@LeaveAllModes
459     \global\let\@gtempa\lst@modestack
460     \global\let\lst@modestack\@empty
461     \expandafter\lst@ReenterModes@\@gtempa\relax
462   \fi}
463 \def\lst@ReenterModes@#1#2{%
464   \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.

```

465     \gdef\@gtempa##1{#1}%
466     \expandafter\@gtempa
467   \else

```

Otherwise we just add the element to `\lst@modestack` and continue the loop.

```
468      \lst@AddTo\lst@modestack{{#1}}%
469      \expandafter\lst@ReenterModes@
470      \fi
471      {#2}}
```

`\lst@LeaveAllModes` Leaving all modes means closing groups until the mode equals `\lst@nomode`.

```
472 \def\lst@LeaveAllModes{%
473   \ifnum\lst@mode=\lst@nomode
474     \expandafter\lsthk@EndGroup
475   \else
476     \expandafter\egroup\expandafter\lst@LeaveAllModes
477   \fi}
```

We need that macro to end a listing correctly.

```
478 \lst@AddToHook{ExitVars}{\lst@LeaveAllModes}
```

`\lst@Pmode` The ‘processing’ and the general purpose mode.

```
\lst@GPmode 479 \lst@NewMode\lst@Pmode
480 \lst@NewMode\lst@GPmode
```

`\lst@modetrue` The usual macro to value a boolean except that we also execute a hook.

```
481 \def\lst@modetrue{\let\lst@ifmode\iftrue \lsthk@ModeTrue}
482 \let\lst@ifmode\iffalse % init
483 \lst@AddToHook{ModeTrue}{}% init
```

`\lst@ifLmode` Comment lines use a static mode. It terminates at end of line.

```
484 \def\lst@Lmodetrue{\let\lst@ifLmode\iftrue}
485 \let\lst@ifLmode\iffalse % init
486 \lst@AddToHook{EOL}{\@whilesw \lst@ifLmode\fi \lst@LeaveMode}
```

13.9 Divers helpers

`\lst@NormedDef` works like `\def` (without any parameters!) but normalizes the replacement text by making all characters lower case and stripping off spaces.

```
487 \def\lst@NormedDef#1#2{\lowercase{\edef#1{\zap@space#2 \@empty}}}
```

`\lst@NormedNameDef` works like `\global\@namedef` (again without any parameters!) but normalizes both the macro name and the replacement text.

```
488 \def\lst@NormedNameDef#1#2{%
489   \lowercase{\edef\lst@temp{\zap@space#1 \@empty}}%
490   \expandafter\xdef\csname\lst@temp\endcsname{\zap@space#2 \@empty}}}
```

`\lst@GetFreeMacro` Initialize `\@tempcnta` and `\lst@freemacro`, ...

```
491 \def\lst@GetFreeMacro#1{%
492   \@tempcnta\z@ \def\lst@freemacro{#1\the\@tempcnta}%
493   \lst@GFM@}
```

... and either build the control sequence or advance the counter and continue.

```
494 \def\lst@GFM@{%
495   \expandafter\ifx \csname\lst@freemacro\endcsname \relax
496     \edef\lst@freemacro{\csname\lst@freemacro\endcsname}%
497   \else
```

```

498         \advance\@tempcnta\@ne
499         \expandafter\lst@GFM@
500     \fi}

```

\lst@gtempboxa

```

501 \newbox\lst@gtempboxa
502 \</kernel>

```

14 Doing output

14.1 Basic registers and keys

```

503 \<kernel>

```

The **current character string** is kept in a token register and a counter holds its length. Here we define the macros to put characters into the output queue.

\lst@token are allocated here. Quite a useful comment, isn't it?
 \lst@length 504 \newtoks\lst@token \newcount\lst@length

\lst@ResetToken The two registers get empty respectively zero at the beginning of each line. After
 \lst@lastother receiving a report from Claus Atzenbeck—I removed such a bug many times—I decided to reset these registers in the EndGroup hook, too.

```

505 \def\lst@ResetToken{\lst@token{}\lst@length\z@}
506 \lst@AddToHook{InitVarsBOL}{\lst@ResetToken \let\lst@lastother\@empty}
507 \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.

```

508 \def\lst@lettertrue{\let\lst@ifletter\iftrue}
509 \def\lst@letterfalse{\let\lst@ifletter\iffalse}
510 \lst@AddToHook{InitVars}{\lst@letterfalse}

```

\lst@Append puts the argument into the output queue.

```

511 \def\lst@Append#1{\advance\lst@length\@ne
512         \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.

```

513 \def\lst@AppendOther{%
514     \lst@ifletter \lst@Output\lst@letterfalse \fi
515     \futurelet\lst@lastother\lst@Append}

```

\lst@AppendLetter We output a non-identifier string if necessary and call \lst@Append.

```

516 \def\lst@AppendLetter{%
517     \lst@ifletter\else \lst@OutputOther\lst@lettertrue \fi
518     \lst@Append}

```


`\lst@SaveToken` If a group end appears and ruins the character string, we can use these macros
`\lst@RestoreToken` to save and restore the contents. `\lst@thestyle` is the current printing style and must be saved and restored, too.

```
519 \def\lst@SaveToken{%
520   \global\let\lst@gthestyle\lst@thestyle
521   \global\let\lst@glastother\lst@lastother
522   \xdef\lst@RestoreToken{\noexpand\lst@token{\the\lst@token}%
523     \noexpand\lst@length\the\lst@length\relax
524     \noexpand\let\noexpand\lst@thestyle
525     \noexpand\lst@gthestyle
526     \noexpand\let\noexpand\lst@lastother
527     \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.

```
528 \def\lst@iflastotheroneof#1{\lst@iflastotheroneof@ #1\relax}
529 \def\lst@iflastotheroneof@#1{%
530   \ifx #1\relax
531     \expandafter\@secondoftwo
532   \else
533     \ifx\lst@lastother#1%
534       \lst@iflastotheroneof@t
535     \else
536       \expandafter\expandafter\expandafter\lst@iflastotheroneof@
537     \fi
538   \fi}
539 \def\lst@iflastotheroneof@t#1\fi\fi#2\relax{\fi\fi\@firstoftwo}
```

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 540 \newdimen\lst@currlwidth % \global
\lst@pos 541 \newcount\lst@column \newcount\lst@pos % \global
542 \lst@AddToHook{InitVarsBOL}
543 {\global\lst@currlwidth\z@ \global\lst@pos\z@ \global\lst@column\z@}
```

`\lst@CalcColumn` sets `\@tempcnta` to the current column. Note that `\lst@pos` will be nonpositive.

```
544 \def\lst@CalcColumn{%
545   \@tempcnta\lst@column
546   \advance\@tempcnta\lst@length
547   \advance\@tempcnta-\lst@pos}
```

`\lst@lostspace` Whenever this dimension is positive we can insert space. A negative ‘lost space’ means that the printed line is wider than expected.

```
548 \newdimen\lst@lostspace % \global
549 \lst@AddToHook{InitVarsBOL}{\global\lst@lostspace\z@}
```

`\lst@UseLostSpace` We insert space and reset it if and only if `\lst@lostspace` is positive.
550 `\def\lst@UseLostSpace{\ifdim\lst@lostspace>\z@ \lst@InsertLostSpace \fi}`

`\lst@InsertLostSpace` Ditto, but insert even if negative. `\lst@Kern` will be defined very soon.
`\lst@InsertHalfLostSpace` 551 `\def\lst@InsertLostSpace{%`
552 `\lst@Kern\lst@lostspace \global\lst@lostspace\z@}`
553 `\def\lst@InsertHalfLostSpace{%`
554 `\global\lst@lostspace.5\lst@lostspace \lst@Kern\lst@lostspace}`

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.

`basewidth` 555 `\newdimen\lst@width`
556 `\lst@Key{basewidth}{0.6em,0.45em}{\lstKV@CSTwoArg{#1}%`
557 `{\def\lst@widthfixed{##1}\def\lst@widthflexible{##2}%`
558 `\ifx\lst@widthflexible\@empty`
559 `\let\lst@widthflexible\lst@widthfixed`
560 `\fi`
561 `\def\lst@temp{\PackageError{Listings}%`
562 `{Negative value(s) treated as zero}%`
563 `\@ehc}%`
564 `\let\lst@error\@empty`
565 `\ifdim \lst@widthfixed<\z@`
566 `\let\lst@error\lst@temp \let\lst@widthfixed\z@`
567 `\fi`
568 `\ifdim \lst@widthflexible<\z@`
569 `\let\lst@error\lst@temp \let\lst@widthflexible\z@`
570 `\fi`
571 `\lst@error}}`

We set the dimension in a special hook.

572 `\lst@AddToHook{FontAdjust}`
573 `{\lst@width=\lst@ifflexible\lst@widthflexible`
574 `\else\lst@widthfixed\fi \relax}`

`fontadjust` This hook is controlled by a switch and is always executed at `InitVars`.

`\lst@FontAdjust` 575 `\lst@Key{fontadjust}{false}[t]{\lstKV@SetIf{#1}\lst@iffontadjust}`
576 `\def\lst@FontAdjust{\lst@iffontadjust \lsthk@FontAdjust \fi}`
577 `\lst@AddToHook{InitVars}{\lsthk@FontAdjust}`

14.2 Low- and mid-level output

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 `\box#1` as argument
`\lst@alloverstyle` to `\lst@alloverstyle`.

578 `\def\lst@OutputBox#1{\lst@alloverstyle{\box#1}}`

Alternative: Instead of `\global\advance\lst@currlwidth\wd{box number}` in both definitions `\lst@Kern` and `\lst@CalcLostSpaceAndOutput`, we could also advance the dimension here. But I decided not to do so since it simplifies possible redefinitions of `\lst@OutputBox`: we need not to care about `\lst@currlwidth`.

```
579 \def\lst@alloverstyle#1{#1}% init
```

\lst@Kern has been used to insert ‘lost space’. It must not use \@tempboxa since that ...

```
580 \def\lst@Kern#1{%
581   \setbox\z@\hbox{{\lst@currstyle{\kern#1}}}%
582   \global\advance\lst@currlwidth \wd\z@
583   \lst@OutputBox\z@}
```

\lst@CalcLostSpaceAndOutput ... is used here. We keep track of \lst@lostspace, \lst@currlwidth and \lst@pos.

```
584 \def\lst@CalcLostSpaceAndOutput{%
585   \global\advance\lst@lostspace \lst@length\lst@width
586   \global\advance\lst@lostspace-\wd\@tempboxa
587   \global\advance\lst@currlwidth \wd\@tempboxa
588   \global\advance\lst@pos -\lst@length
```

Before \@tempboxa is output, we insert space if there is enough lost space. This possibly invokes \lst@Kern via ‘insert half lost space’, which is the reason for why we mustn’t use \@tempboxa above. By redefinition we prevent \lst@OutputBox from using any special style in \lst@Kern.

```
589   \setbox\@tempboxa\hbox{\let\lst@OutputBox\box
590     \ifdim\lst@lostspace>\z@ \lst@leftinsert \fi
591     \box\@tempboxa
592     \ifdim\lst@lostspace>\z@ \lst@rightinsert \fi}}%
```

Finally we can output the new box.

```
593   \lst@OutputBox\@tempboxa \lsthk@PostOutput}
594 \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.

```
595 \def\lst@OutputToken{%
596   \lst@TrackNewLines \lst@OutputLostSpace
597   \lst@ifgobbledws
598     \lst@gobbledwhitespacefalse
599     \lst@@discretionary
600   \fi
601   \lst@CheckMerge
602   {\lst@thestyle{\lst@FontAdjust
603     \setbox\@tempboxa\lst@hbox
604     {\lsthk@OutputBox
605       \lst@lefthss
606       \expandafter\lst@FillOutputBox\the\lst@token\@empty
607       \lst@righthss}}%
608   \lst@CalcLostSpaceAndOutput}}%
609   \lst@ResetToken}

610 \lst@AddToHook{OutputBox}{}% init

611 \def\lst@gobbledwhitespacetrue{\global\let\lst@ifgobbledws\iftrue}
612 \def\lst@gobbledwhitespacefalse{\global\let\lst@ifgobbledws\iffalse}
613 \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` To delay or merge #1, we process it as usual and simply save the state in macros.
`\lst@Merge` For delayed characters we also need the currently ‘active’ output routine. Both definitions first check whether there are already delayed or ‘merged’ characters.

```

614 \def\lst@Delay#1{%
615     \lst@CheckDelay
616     #1%
617     \lst@GetOutputMacro\lst@delayedoutput
618     \edef\lst@delayed{\the\lst@token}%
619     \edef\lst@delayedlength{\the\lst@length}%
620     \lst@ResetToken}

621 \def\lst@Merge#1{%
622     \lst@CheckMerge
623     #1%
624     \edef\lst@merged{\the\lst@token}%
625     \edef\lst@mergedlength{\the\lst@length}%
626     \lst@ResetToken}

```

`\lst@MergeToken` Here we put the things together again.

```

627 \def\lst@MergeToken#1#2{%
628     \advance\lst@length#2%
629     \lst@lExtend#1{\the\lst@token}%
630     \expandafter\lst@token\expandafter{#1}%
631     \let#1\@empty}

```

`\lst@CheckDelay` 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.

```

632 \def\lst@CheckDelay{%
633     \ifx\lst@delayed\@empty\else
634         \lst@GetOutputMacro\@gtempa
635         \ifx\lst@delayedoutput\@gtempa
636             \lst@MergeToken\lst@delayed\lst@delayedlength
637         \else
638             {\lst@ResetToken
639             \lst@MergeToken\lst@delayed\lst@delayedlength
640             \lst@delayedoutput}%
641             \let\lst@delayed\@empty
642         \fi
643     \fi}

```

`\lst@CheckMerge` All this is easier for `\lst@merged`.

```

644 \def\lst@CheckMerge{%
645     \ifx\lst@merged\@empty\else
646         \lst@MergeToken\lst@merged\lst@mergedlength
647     \fi}

```

```
648 \let\lst@delayed\@empty % init
649 \let\lst@merged\@empty % init
```

14.3 Column formats

It's time to deal with fixed and flexible column modes. A couple of open definitions are now filled in.

`\lst@column@fixed` switches to the fixed column format. The definitions here control how the output of the above definitions looks like.

```
650 \def\lst@column@fixed{%
651   \lst@flexiblefalse
652   \lst@width\lst@widthfixed\relax
653   \let\lst@OutputLostSpace\lst@UseLostSpace
654   \let\lst@FillOutputBox\lst@FillFixed
655   \let\lst@hss\hss
656   \def\lst@hbox{\hbox to\lst@length\lst@width}}
```

`\lst@FillFixed` Filling up a fixed mode box is easy.

```
657 \def\lst@FillFixed#1{#1\lst@FillFixed@}
```

While not reaching the end (`\@empty` from above), we insert dynamic space, output the argument and call the submacro again.

```
658 \def\lst@FillFixed@#1{%
659   \ifx\@empty#1\else \lst@hss#1\expandafter\lst@FillFixed@ \fi}
```

`\lst@column@flexible` The first flexible format.

```
660 \def\lst@column@flexible{%
661   \lst@flexibletrue
662   \lst@width\lst@widthflexible\relax
663   \let\lst@OutputLostSpace\lst@UseLostSpace
664   \let\lst@FillOutputBox\@empty
665   \let\lst@hss\@empty
666   \let\lst@hbox\hbox}
```

`\lst@column@fullflexible` This column format inserts no lost space except at the beginning of a line.

```
667 \def\lst@column@fullflexible{%
668   \lst@column@flexible
669   \def\lst@OutputLostSpace{\lst@ifnewline \lst@UseLostSpace\fi}%
670   \let\lst@leftinsert\@empty
671   \let\lst@rightinsert\@empty}
```

So far 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`.

```
672 \def\lst@outputpos#1#2\relax{%
673   \def\lst@lefthss{\lst@hss}\let\lst@righthss\lst@lefthss
674   \let\lst@rightinsert\lst@InsertLostSpace
675   \ifx #1c%
676     \let\lst@leftinsert\lst@InsertHalfLostSpace
677   \else\ifx #1r%
```

```

678     \let\lst@righthss\@empty
679     \let\lst@leftinsert\lst@InsertLostSpace
680     \let\lst@rightinsert\@empty
681   \else
682     \let\lst@lefthss\@empty
683     \let\lst@leftinsert\@empty
684     \ifx #1\else \PackageWarning{Listings}%
685       {Unknown positioning for output boxes}%
686     \fi
687   \fi\fi}

```

\lst@ifflexible indicates the column mode but does not distinguish between different fixed or flexible modes.

```

688 \def\lst@flexibletrue{\let\lst@ifflexible\iftrue}
689 \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.

```

690 \lst@Key{columns}{[c]fixed}{\lstKV@OptArg[] {#1}{%
691   \ifx\@empty##1\@empty\else \lst@outputpos##1\relax\relax \fi
692   \expandafter\let\expandafter\lst@arg
693     \csname\@lst @column@##2\endcsname
694   We issue a warning or save the definition for later.
695   \lst@arg
696   \ifx\lst@arg\relax
697     \PackageWarning{Listings}{Unknown column format ‘##2’}%
698   \else
699     \lst@ifflexible
700     \let\lst@columnsflexible\lst@arg
701   \else
702     \let\lst@columnsfixed\lst@arg
703   \fi
704   \fi}}
705 \let\lst@columnsfixed\lst@column@fixed % init
706 \let\lst@columnsflexible\lst@column@flexible % init

```

flexiblecolumns Nothing else but a key to switch between the last flexible and fixed mode.

```

706 \lst@Key{flexiblecolumns}\relax[t]{%
707   \lstKV@SetIf{#1}\lst@ifflexible
708   \lst@ifflexible \lst@columnsflexible
709   \else \lst@columnsfixed \fi}

```

14.4 New lines

\lst@newlines This counter holds the number of ‘new lines’ (cr+lf) we have to perform.

```

710 \newcount\lst@newlines
711 \lst@AddToHook{InitVars}{\global\lst@newlines\z@}
712 \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.

```

713 \def\lst@NewLine{%
714     \ifx\lst@OutputBox\@gobble\else
715         \par\noindent \hbox{}%
716     \fi
717     \global\advance\lst@newlines\m@ne
718     \lst@newlinetrue}

```

Define \lst@newlinetrue and reset if after output.

```

719 \def\lst@newlinetrue{\global\let\lst@ifnewline\iftrue}
720 \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.

```

721 \def\lst@TrackNewLines{%
722     \ifnum\lst@newlines>\z@
723         \lsthk@OnNewLine
724         \lst@DoNewLines
725     \fi}
726 \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.

```

727 \lst@Key{emptylines}\maxdimen{%
728     \ifstar{\lst@true\@tempcnta\@gobble#1\relax\lst@GobbleNil}%
729         {\lst@false\@tempcnta#1\relax\lst@GobbleNil}#1\@nil
730     \advance\@tempcnta\@ne
731     \edef\lst@maxempty{\the\@tempcnta\relax}%
732     \let\lst@ifpreservernumber\lst@if}

```

\lst@DoNewLines First we take care of \lst@maxempty and then of the remaining empty lines.

```

733 \def\lst@DoNewLines{
734     \@whilenum\lst@newlines>\lst@maxempty \do
735         {\lst@ifpreservernumber
736             \lsthk@OnEmptyLine
737             \global\advance\c@lstnumber\lst@advancelstnum
738         \fi
739         \global\advance\lst@newlines\m@ne}%
740     \@whilenum \lst@newlines>\@ne \do
741         {\lsthk@OnEmptyLine \lst@NewLine}%
742     \ifnum\lst@newlines>\z@ \lst@NewLine \fi}
743 \lst@AddToHook{OnEmptyLine}{}% init

```

14.5 High-level output

identifierstyle A simple key.

```

744 \lst@Key{identifierstyle}{\def\lst@identifierstyle{#1}}
745 \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.

```

746 \def\lst@GotoTabStop{%
747     \ifnum\lst@newlines>\z@
748         \setbox\@tempboxa\hbox{\lst@outputspace}%
749         \setbox\@tempboxa\hbox to\wd\@tempboxa{{\lst@currstyle{\hss}}}%
750         \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.

```
751 \else
```

Otherwise (no printed characters) we only need to advance `\lst@lostspace`, which is inserted by `\lst@OutputToken` above, and update the column.

```
752 \global\advance\lst@lostspace \lst@length\lst@width
753 \global\advance\lst@column\lst@length \lst@length\z@
754 \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.

```
755 \def\lst@OutputOther{%
756 \lst@CheckDelay
757 \ifnum\lst@length=\z@\else
758 \let\lst@thestyle\lst@currstyle
759 \lsthk@OutputOther
760 \lst@OutputToken
761 \fi}
762 \lst@AddToHook{OutputOther}{}% init
763 \let\lst@currstyle\relax % init
```

`\lst@Output` We might use identifier style as default.

```
764 \def\lst@Output{%
765 \lst@CheckDelay
766 \ifnum\lst@length=\z@\else
767 \ifx\lst@currstyle\relax
768 \let\lst@thestyle\lst@identifierstyle
769 \else
770 \let\lst@thestyle\lst@currstyle
771 \fi
772 \lsthk@Output
773 \lst@OutputToken
774 \fi
775 \let\lst@lastother\relax}
```

Note that `\lst@lastother` becomes equivalent to `\relax` and not equivalent to `\@empty` as everywhere else. I don't know whether this will be important in the future or not.

```
776 \lst@AddToHook{Output}{}% init
```

`\lst@GetOutputMacro` Just saves the output macro to be used.

```
777 \def\lst@GetOutputMacro#1{%
778 \lst@ifletter \global\let#1\lst@Output
779 \else \global\let#1\lst@OutputOther\fi}
```

`\lst@PrintToken` outputs the current character string in letter or nonletter mode.

```
780 \def\lst@PrintToken{%
781 \lst@ifletter \lst@Output \lst@letterfalse
782 \else \lst@OutputOther \let\lst@lastother\@empty \fi}
```


`\lst@XPrintToken` is a special definition to print also merged characters.

```
783 \def\lst@XPrintToken{%
784   \lst@PrintToken \lst@CheckMerge
785   \ifnum\lst@length=\z@\else \lst@PrintToken \fi}
```

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

```
786 \def\lst@BeginDropOutput#1{%
787   \xdef\lst@BDOnewlines{\the\lst@newlines}%
788   \global\let\lst@BDOnewline\lst@ifnewline
789   \lst@EnterMode{#1}%
790   {\lst@modetrue
791     \let\lst@OutputBox\@gobble
792     \aftergroup\lst@BDORestore}}
```

Restoring the data is quite easy:

```
793 \def\lst@BDORestore{%
794   \global\lst@newlines\lst@BDOnewlines
795   \global\let\lst@ifnewline\lst@BDOnewline}
```

`\lst@EndDropOutput` is equivalent to `\lst@LeaveMode`.

```
796 \let\lst@EndDropOutput\lst@LeaveMode
797 </kernel>
```

14.7 Writing to an external file

Now it would be good to know something about character classes since we need to access the true input characters, for example a tabulator and not the spaces it 'expands' to.

```
798 <*misc>
799 \lst@BeginAspect{writefile}
```

`\lst@WF` The contents of the token will be written to file.

```
\lst@WFtoken 800 \newtoks\lst@WFtoken % global
801 \lst@AddToHook{InitVarsBOL}{\global\lst@WFtoken{}}
802 \newwrite\lst@WF
803 \global\let\lst@WFifopen\iffalse % init
```

`\lst@WFWriteToFile` To do this, we have to expand the contents and then expand this via `\edef`. Empty `\lst@UM` ensures that special characters (underscore, dollar, etc.) are written correctly.

```
804 \gdef\lst@WFWriteToFile{%
805   \begingroup
806   \let\lst@UM\@empty
807   \expandafter\edef\expandafter\lst@temp\expandafter{\the\lst@WFtoken}%
808   \immediate\write\lst@WF{\lst@temp}%
809   \endgroup
810   \global\lst@WFtoken{}}
```

```

\lst@WFAppend Similar to \lst@Append but uses \lst@WFToken.
811 \gdef\lst@WFAppend#1{%
812     \global\lst@WFToken=\expandafter{\the\lst@WFToken#1}}

\lst@BeginWriteFile use different macros for \lst@OutputBox (not) to drop the output.
\lst@BeginAlsoWriteFile 813 \gdef\lst@BeginWriteFile{\lst@WFBEGIN\@gobble}
814 \gdef\lst@BeginAlsoWriteFile{\lst@WFBEGIN\lst@OutputBox}

\lst@WFBEGIN Here ...
815 \begingroup \catcode'\^^I=11
816 \gdef\lst@WFBEGIN#1#2{%
817     \begingroup
818     \let\lst@OutputBox#1%
    ... we have to update \lst@WFToken and ...
819     \def\lst@Append##1{%
820         \advance\lst@length\@ne
821         \expandafter\lst@token\expandafter{\the\lst@token##1}%
822         \ifx ##1\lst@outputspace \else
823             \lst@WFAppend##1%
824         \fi}%
825     \lst@lAddTo\lst@PreGotoTabStop{\lst@WFAppend{\^^I}}%
826     \lst@lAddTo\lst@ProcessSpace{\lst@WFAppend{ }}%
    ... need different 'EOL' and 'DeInit' definitions to write the token register to file.
827     \let\lst@DeInit\lst@WFDeInit
828     \let\lst@MProcessListing\lst@WFMProcessListing
    Finally we open the file if necessary.
829     \lst@WFifopen\else
830         \immediate\openout\lst@WF=#2\relax
831         \global\let\lst@WFifopen\iftrue
832         \@gobbletwo\fi\fi
833     \fi}
834 \endgroup

\lst@EndWriteFile closes the file and restores original definitions.
835 \gdef\lst@EndWriteFile{%
836     \immediate\closeout\lst@WF \endgroup
837     \global\let\lst@WFifopen\iffalse}

\lst@WFMProcessListing write additionally \lst@WFToken to external file.
\lst@WFDeInit 838 \global\let\lst@WFMProcessListing\lst@MProcessListing
839 \global\let\lst@WFDeInit\lst@DeInit
840 \lst@AddToAtTop\lst@WFMProcessListing{\lst@WFWriteToFile}
841 \lst@AddToAtTop\lst@WFDeInit{%
842     \ifnum\lst@length=\z@\else \lst@WFWriteToFile \fi}

843 \lst@EndAspect
844 \</misc>

```

15 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

```

845 \kernel
\lst@ProcessLetter We put the letter, which is not a whitespace, into the output queue.
846 \def\lst@ProcessLetter{\lst@whitespacefalse \lst@AppendLetter}

\lst@ProcessOther Ditto.
847 \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.
848 \def\lst@ProcessDigit{%
849     \lst@whitespacefalse
850     \lst@ifletter \expandafter\lst@AppendLetter
851     \else \expandafter\lst@AppendOther\fi}

\lst@ifwhitespace indicates whether the last processed character has been white space.
852 \def\lst@whitespacetrue{\global\let\lst@ifwhitespace\iftrue}
853 \def\lst@whitespacefalse{\global\let\lst@ifwhitespace\iffalse}
854 \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 Niepraschk.
855 \lst@Key{tabsize}{8}
856     {\ifnum#1>\z@ \def\lst@tabsize{#1}\else
857         \PackageError{Listings}{Strict positive integer expected}%
858         {You can't use '#1' as tabsize. \@ehc}%
859     \fi}

showtabs Two more user keys for tab control.
tab 860 \lst@Key{showtabs}f[t]{\lstKV@SetIf{#1}\lst@ifshowtabs}
861 \lst@Key{tab}{\kern.06em\hbox{\vrule\@height.3ex}%
862     \hrulefill\hbox{\vrule\@height.3ex}}
863     {\def\lst@tab{#1}}

```

`\lst@ProcessTabulator` A tabulator outputs the preceding characters, which decrements `\lst@pos` by the number of printed characters.

```
864 \def\lst@ProcessTabulator{%
865     \lst@XPrintToken \lst@whitespacetrue

Then we calculate how many columns we need to reach the next tabulator stop:
we add \lst@tabsize until \lst@pos is strict positive. In other words, \lst@pos
is the column modulo tabsize and we're looking for a positive representative. We
assign it to \lst@length and reset \lst@pos in the submacro.

866     \global\advance\lst@column -\lst@pos
867     \@whilenum \lst@pos<\@ne \do
868         {\global\advance\lst@pos\lst@tabsize}%
869     \lst@length\lst@pos
870     \lst@PreGotoTabStop}
```

`\lst@PreGotoTabStop` Visible tabs print `\lst@tab`.

```
871 \def\lst@PreGotoTabStop{%
872     \lst@ifshowtabs
873         \lst@TrackNewLines
874         \setbox\@tempboxa\hbox to\lst@length\lst@width
875             {\lst@currstyle{\hss\lst@tab}}}%
876         \lst@CalcLostSpaceAndOutput
877     \else

If we are advised to keep spaces, we insert the correct number of them.

878         \lst@ifkeepspace
879             \@tempcnta\lst@length \lst@length\z@
880             \@whilenum \@tempcnta>\z@ \do
881                 {\lst@AppendOther\lst@outputspace
882                     \advance\@tempcnta\m@ne}%
883             \lst@OutputOther
884         \else
885             \lst@GotoTabStop
886         \fi
887     \fi
888     \lst@length\z@ \global\lst@pos\z@}
```

Spaces are implemented as described at the beginning of this subsection. But first we define some user keys.

`\lst@outputspace` The first macro is a default definition, ...

```
\lst@visiblepace 889 \def\lst@outputspace{\ }
890 \def\lst@visiblepace{\lst@ttfamily{\char32}\textvisiblepace}
```

`showspaces` ... which is modified on user's request.

```
keepspace 891 \lst@Key{showspaces}{false}[t]{\lstKV@SetIf{#1}\lst@ifshowspaces}
892 \lst@Key{keepspace}{false}[t]{\lstKV@SetIf{#1}\lst@ifkeepspace}
893 \lst@AddToHook{Init}
894     {\lst@ifshowspaces
895         \let\lst@outputspace\lst@visiblepace
896         \lst@keepspacestrue
897     \fi}
898 \def\lst@keepspacestrue{\let\lst@ifkeepspace\iftrue}
```

`\lst@ProcessSpace` We look whether spaces fix the column alignment or not. In the latter case we append a space; otherwise ...

```

899 \def\lst@ProcessSpace{%
900     \lst@ifkeepspace
901         \lst@whitespace true
902         \lst@PrintToken
903         \lst@AppendOther\lst@outputspace
904         \lst@PrintToken
905     \else \ifnum\lst@newlines=\z@
... we append a ‘special space’ if the line isn’t empty.
906         \lst@AppendSpecialSpace
907     \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.

```

908         \global\advance\lst@lostspace\lst@width
909         \global\advance\lst@pos\m@ne
910         \lst@whitespace true
911     \else
912         \lst@AppendSpecialSpace
913     \fi
914 \fi \fi}

```

Note that this version works for fixed and flexible column output.

`\lst@AppendSpecialSpace` 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.

```

915 \def\lst@AppendSpecialSpace{%
916     \lst@ifwhitespace
917         \lst@PrintToken
918         \global\advance\lst@lostspace\lst@width
919         \global\advance\lst@pos\m@ne
920         \lst@gobbledwhitespace true
921     \else
922         \lst@whitespace true
923         \lst@PrintToken
924         \lst@AppendOther\lst@outputspace
925         \lst@PrintToken
926     \fi}

```

Form feeds has been introduced after communication with Jan Braun.

`formfeed` let the user make adjustments.

```

927 \lst@Key{formfeed}{\bigbreak}{\def\lst@formfeed{#1}}

```

`\lst@ProcessFormFeed` 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.

```

928 \def\lst@ProcessFormFeed{%
929     \lst@XPrintToken
930     \ifnum\lst@newlines=\z@

```

```

931      \lst@EOLUpdate \lsthk@InitVarsBOL
932      \fi
933      \lst@formfeed
934      \lst@whitespace>true}

```

15.3 Character tables

15.3.1 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` $\langle class macro \rangle \langle c_1 \rangle \dots \langle c_k \rangle \backslash z @$

extends the standard character table by the characters with codes $\langle c_1 \rangle \dots \langle c_k \rangle$ making each character use $\langle class macro \rangle$. All these characters must be printable via `\char` $\langle c_i \rangle$.

`\lst@CCPutMacro` $\langle class_1 \rangle \langle c_1 \rangle \langle definition_1 \rangle \dots \backslash @empty \backslash z @ \backslash @empty$

also extends the standard character table: the character $\langle c_i \rangle$ will use $\langle class_i \rangle$ and is printed via $\langle definition_i \rangle$. These definitions must be *spec. token*s in the sense of section 9.5.

`\lst@Def` For speed we won't use these helpers too often.

```

\lst@Let 935 \def\lst@Def#1{\lccode'\~=#1\lowercase{\def~}}
          936 \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.

```

937 \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.

```

938 \def\lst@SelectStdCharTable{%
939     \lst@Def{9}{\lst@ProcessTabulator}%
940     \lst@Def{12}{\lst@ProcessFormFeed}%
941     \lst@Def{32}{\lst@ProcessSpace}}

```

`\lst@CCPut` The first argument gives the character class, then follow the codes.

```

942 \def\lst@CCPut#1#2{%
943     \ifnum#2=\backslash z @
944         \expandafter\gobbletwo
945     \else
946         \lccode'\~=#2\lccode'\/= #2\lowercase{\lst@CCPut@~{#1/}}%
947     \fi
948     \lst@CCPut#1}
949 \def\lst@CCPut@#1#2{\lst@lAddTo\lst@SelectStdCharTable{\def#1{#2}}}

```

Now we insert more standard characters.

```

950 \lst@CCPut \lst@ProcessOther
951     {"21"}{"22"}{"28"}{"29"}{"2B"}{"2C"}{"2E"}{"2F"}
952     {"3A"}{"3B"}{"3D"}{"3F"}{"5B"}{"5D"}
953     \z@
954 \lst@CCPut \lst@ProcessDigit
955     {"30"}{"31"}{"32"}{"33"}{"34"}{"35"}{"36"}{"37"}{"38"}{"39"}
956     \z@
957 \lst@CCPut \lst@ProcessLetter
958     {"40"}{"41"}{"42"}{"43"}{"44"}{"45"}{"46"}{"47"}
959     {"48"}{"49"}{"4A"}{"4B"}{"4C"}{"4D"}{"4E"}{"4F"}
960     {"50"}{"51"}{"52"}{"53"}{"54"}{"55"}{"56"}{"57"}
961     {"58"}{"59"}{"5A"}
962     {"61"}{"62"}{"63"}{"64"}{"65"}{"66"}{"67"}
963     {"68"}{"69"}{"6A"}{"6B"}{"6C"}{"6D"}{"6E"}{"6F"}
964     {"70"}{"71"}{"72"}{"73"}{"74"}{"75"}{"76"}{"77"}
965     {"78"}{"79"}{"7A"}
966     \z@

```

`\lst@CCPutMacro` Now we come to a delicate point. The characters not inserted yet aren't printable (`_`, `$`, ...) or aren't printed well (`*`, `-`, ...) 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.

```

967 \def\lst@CCPutMacro#1#2#3{%
968     \ifnum#2=\z@ \else
969         \begingroup\lccode'\~=#2\relax \lccode'\/= #2\relax
970         \lowercase{\endgroup\expandafter\lst@CCPutMacro@
971             \csname\@lst @um/\expandafter\endcsname
972             \csname\@lst @um/@\endcsname /\~}#1{#3}%
973         \expandafter\lst@CCPutMacro
974     \fi}

```

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 _}` and `\lst@um_@`.

```

975 \def\lst@CCPutMacro@#1#2#3#4#5#6{%
976     \lst@lAddTo\lst@SelectStdCharTable{\def#4{#5#1}}%
977     \def#1{\lst@UM#3}%
978     \def#2{#6}}

```

The default definition of `\lst@UM`:

```
979 \def\lst@UM#1{\csname\@lst @um#1@\endcsname}
```

And all remaining standard characters.

```
980 \lst@CCPutMacro
981   \lst@ProcessOther {"23}\#
982   \lst@ProcessLetter{"24}\textdollar
983   \lst@ProcessOther {"25}\%
984   \lst@ProcessOther {"26}\&
985   \lst@ProcessOther {"27}{\lst@ifupquote \textquotesingle
986                               \else \char39\relax \fi}
987   \lst@ProcessOther {"2A}{\lst@ttfamily*\textasteriskcentered}
988   \lst@ProcessOther {"2D}{\lst@ttfamily{-}}{\$-\$}}
989   \lst@ProcessOther {"3C}{\lst@ttfamily<\textless}
990   \lst@ProcessOther {"3E}{\lst@ttfamily>\textgreater}
991   \lst@ProcessOther {"5C}{\lst@ttfamily{\char92}\textbackslash}
992   \lst@ProcessOther {"5E}\textasciicircum
993   \lst@ProcessLetter{"5F}{\lst@ttfamily{\char95}\textunderscore}
994   \lst@ProcessOther {"60}{\lst@ifupquote \textasciigrave
995                               \else \char96\relax \fi}
996   \lst@ProcessOther {"7B}{\lst@ttfamily{\char123}\textbraceleft}
997   \lst@ProcessOther {"7C}{\lst@ttfamily|\textbar}
998   \lst@ProcessOther {"7D}{\lst@ttfamily{\char125}\textbraceright}
999   \lst@ProcessOther {"7E}\textasciitilde
1000  \lst@ProcessOther {"7F}-
1001  \@empty\z@\@empty
```

`\lst@ttfamily` What is this ominous macro? It prints either the first or the second argument. In `\ttfamily` 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

```
1002 \def\lst@ttfamily#1#2{\ifx\family\ttdefault#1\relax\else#2\fi}
    \ttdefault is defined \long, so the \ifx doesn't work since \family isn't
    \long! We go around this problem by redefining \ttdefault locally:
1003 \lst@AddToHook{Init}{\edef\ttdefault{\ttdefault}}
```

`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.

```
1004 \lst@Key{upquote}{false}[t]{\lstKV@SetIf{#1}\lst@ifupquote
1005   \lst@ifupquote
1006     \@ifundefined{textasciigrave}%
1007       {\let\KV@lst@upquote\@gobble
1008         \lstKV@SetIf f\lst@ifupquote \@gobble\fi
1009         \PackageError{Listings}{Option 'upquote' requires 'textcomp'
1010           package.\MessageBreak The option has been disabled}%
1011         {Add \string\usepackage{textcomp} to your preamble.}}%
1012       {}%
1013   \fi}
```

If an `upquote` package is loaded, the `upquote` option is enabled by default.

```
1014 \AtBeginDocument{%
1015   \@ifpackageloaded{upquote}{\RequirePackage{textcomp}}%
```



```

1016 \lstset{upquote}}{}%
1017 \@ifpackageloaded{upquote2}{\lstset{upquote}}{}}

```

`\lst@ifactivechars` A simple switch.

```

1018 \def\lst@activecharstrue{\let\lst@ifactivechars\iftrue}
1019 \def\lst@activecharsfalse{\let\lst@ifactivechars\iffalse}
1020 \lst@activecharstrue

```

`\lst@SelectCharTable` We select the standard character table and switch to active catcodes.

```

1021 \def\lst@SelectCharTable{%
1022   \lst@SelectStdCharTable
1023   \lst@ifactivechars
1024     \catcode9\active \catcode12\active \catcode13\active
1025     \@tempcnta=32\relax
1026     \@whilenum\@tempcnta<128\do
1027       {\catcode\@tempcnta\active\advance\@tempcnta\@ne}%
1028   \fi
1029   \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.

```

1030 \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.

```

1031 \lsthk@SelectCharTable
1032 \lst@DeveloperSCT
1033 \ifx\lst@Backslash\relax\else
1034   \lst@LetSaveDef{"5C}\lsts@backslash\lst@Backslash
1035 \fi}

```

`SelectCharTable` The keys to adjust `\lst@DeveloperSCT`.

```

MoreSelectCharTable 1036 \lst@Key{SelectCharTable}{}{\def\lst@DeveloperSCT{#1}}
1037 \lst@Key{MoreSelectCharTable}\relax{\lst@lAddTo\lst@DeveloperSCT{#1}}
1038 \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` for certain characters. Note that the submacro is a special kind of a local `\lst@AddToAtTop`. The submacro definition was fixed thanks to Peter Bartke.

```

1039 \def\lst@do@noligs#1{%
1040   \begingroup \lccode'\~='#1\lowercase{\endgroup
1041   \lst@do@noligs@~}}
1042 \def\lst@do@noligs@#1{%
1043   \expandafter\expandafter\expandafter\def
1044   \expandafter\expandafter\expandafter#1%
1045   \expandafter\expandafter\expandafter{\expandafter\lst@NoLig#1}}

```

`\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.

```

1046 \def\lst@NoLig{\advance\lst@length\m@ne \lst@Append\lst@nolig}
1047 \def\lst@nolig{\lst@UM\@empty}%

```

But the usual meaning of `\lst@UM` builds the following control sequence, which prevents ligatures in the manner of L^AT_EX's `\do@noligs`.

```

1048 \@namedef{\@lst @um@}{\leavevmode\kern\z@}

```

`\lst@SaveOutputDef` To get the *spec. token* meaning of character #1, we look for `\def` ‘active character #1’ in `\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.

```

1049 \def\lst@SaveOutputDef#1#2{%
1050   \begingroup \lccode'\~=#1\relax \lowercase{\endgroup
1051   \def\lst@temp##1\def~##2##3\relax}{%
1052     \global\expandafter\let\expandafter#2\@gobble##2\relax}%
1053   \expandafter\lst@temp\lst@SelectStdCharTable\relax}

```

`\lstum@backslash` A commonly used character.

```

1054 \lst@SaveOutputDef{"5C}\lstum@backslash

```

15.3.2 National characters

`extendedchars` The user key to activate extended characters 128–255.

```

1055 \lst@Key{extendedchars}{false}[t]{\lstKV@SetIf{#1}\lst@ifec}

```

`\lst@DefEC` Currently each character in the range 128–255 is treated as a letter.

```

1056 \def\lst@DefEC{%
1057   \lst@CCECUse \lst@ProcessLetter
1058   ^^80^^81^^82^^83^^84^^85^^86^^87^^88^^89^^8a^^8b^^8c^^8d^^8e^^8f%
1059   ^^90^^91^^92^^93^^94^^95^^96^^97^^98^^99^^9a^^9b^^9c^^9d^^9e^^9f%
1060   ^^a0^^a1^^a2^^a3^^a4^^a5^^a6^^a7^^a8^^a9^^aa^^ab^^ac^^ad^^ae^^af%
1061   ^^b0^^b1^^b2^^b3^^b4^^b5^^b6^^b7^^b8^^b9^^ba^^bb^^bc^^bd^^be^^bf%
1062   ^^c0^^c1^^c2^^c3^^c4^^c5^^c6^^c7^^c8^^c9^^ca^^cb^^cc^^cd^^ce^^cf%
1063   ^^d0^^d1^^d2^^d3^^d4^^d5^^d6^^d7^^d8^^d9^^da^^db^^dc^^dd^^de^^df%
1064   ^^e0^^e1^^e2^^e3^^e4^^e5^^e6^^e7^^e8^^e9^^ea^^eb^^ec^^ed^^ee^^ef%
1065   ^^f0^^f1^^f2^^f3^^f4^^f5^^f6^^f7^^f8^^f9^^fa^^fb^^fc^^fd^^fe^^ff%
1066   ^^00}

```

`\lst@CCECUse` Reaching end of list (`^^00`) we terminate the loop. Otherwise we do the same as in `\lst@CCPut` if the character is not active. But if the character is active, we save the meaning before redefinition.

```

1067 \def\lst@CCECUse#1#2{%
1068   \ifnum'#2=\z@
1069     \expandafter\@gobbletwo
1070   \else
1071     \ifnum\catcode'#2=\active
1072       \lccode'\~='#2\lccode'\/'='#2\lowercase{\lst@CCECUse@#1~/}%
1073     \else
1074       \lst@ifactivechars \catcode'#2=\active \fi
1075       \lccode'\~='#2\lccode'\/'='#2\lowercase{\def~{#1/}}%
1076     \fi
1077   \fi
1078   \lst@CCECUse#1}

```

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.

```
1079 \def\lst@CCECUse@#1#2#3{%
1080   \expandafter\def\csname\@lst @EC#3\endcsname{\lst@UM#3}%
1081   \expandafter\let\csname\@lst @um#3\endcsname #2%
1082   \edef#2{\noexpand#1%
1083     \expandafter\noexpand\csname\@lst @EC#3\endcsname}}
```

Daniel Gerigk and Heiko Oberdiek reported an error and a solution, respectively.

15.3.3 Catcode problems

\lst@nfss@catcodes 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.

```
1084 \lst@AddToHook{Init}
1085   {\let\lsts@nfss@catcodes\nfss@catcodes
1086    \let\nfss@catcodes\lst@nfss@catcodes}
```

The &-character had turned into \& after a bug report by David Aspinall.

```
1087 \def\lst@nfss@catcodes{%
1088   \lst@makeletter
1089   ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz\relax
1090   \@makeother (\@makeother )\@makeother ,\@makeother :\@makeother \&%
1091   \@makeother 0\@makeother 1\@makeother 2\@makeother 3\@makeother 4%
1092   \@makeother 5\@makeother 6\@makeother 7\@makeother 8\@makeother 9%
1093   \@makeother =\lsts@nfss@catcodes}
```

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 \@makeaoother (.

The name of \lst@makeletter is an imitation of L^AT_EX’s \@makeother.

```
1094 \def\lst@makeletter#1{%
1095   \ifx\relax#1\else\catcode'#111\relax \expandafter\lst@makeletter\fi}
```

\output Another problem was first reported by Marcin Kasperski. It is also catcode related and Donald Arseneau let me understand it. The point is that T_EX seems to use the *currently* active catcode table when it writes non-\immediate \writes to file and not the catcodes involved when *reading* the characters. So a section heading \L a 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.

The solution is a local redefinition of the output routine. 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 T_EX 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 backed_up,

which prevents \TeX 's from recovering from an unbalanced output routine. Heiko Oberdiek reported that `\csname egroup\endcsname` does the trick, too.

However, since \TeX checks the contents of `\box 255` when we close the group ('output routine didn't use all of `\box 255`'), we have to save it temporarily.

```
1096 \lst@AddToHook{Init}
1097 {\edef\lst@OrgOutput{\the\output}%
1098 \output{\global\setbox\lst@gtempboxa\box\@cclv
1099 \expandafter\egroup
```

Now we can interrupt the mode, but we have to save the current character string and the current style.

```
1100 \lst@SaveToken
1101 \lst@InterruptModes
```

We restore the contents, use the original output routine, and ...

```
1102 \setbox\@cclv\box\lst@gtempboxa
1103 \bgroup\lst@OrgOutput\egroup
```

... open a group matching the `}` which \TeX inserts at the end of the output routine. We reenter modes and restore the character string and style `\aftergroup`. Moreover we need to reset `\pagegoal`—added after a bug report by Jochen Schneider.

```
1104 \bgroup
1105 \aftergroup\pagegoal\aftergroup\vsizer
1106 \aftergroup\lst@ReenterModes\aftergroup\lst@RestoreToken}}
```

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.

15.3.4 Adjusting the table

We begin with modifiers for the basic character classes.

alsoletter The macros `\lst@also...` will hold `\def<char>{...}` sequences, which adjusts **alsodigit** the standard character table.

```
alsoother 1107 \lst@Key{alsoletter}\relax{%
1108 \lst@DoAlso{#1}\lst@alsoletter\lst@ProcessLetter}
1109 \lst@Key{alsodigit}\relax{%
1110 \lst@DoAlso{#1}\lst@alsodigit\lst@ProcessDigit}
1111 \lst@Key{alsoother}\relax{%
1112 \lst@DoAlso{#1}\lst@alsoother\lst@ProcessOther}
```

This is done at `SelectCharTable` and every language selection the macros get empty.

```
1113 \lst@AddToHook{SelectCharTable}
1114 {\lst@alsoother \lst@alsodigit \lst@alsoletter}
1115 \lst@AddToHookExe{SetLanguage}% init
1116 {\let\lst@alsoletter\@empty
1117 \let\lst@alsodigit\@empty
1118 \let\lst@alsoother\@empty}
```

The service macro starts a loop and ...

```
1119 \def\lst@DoAlso#1#2#3{%
1120 \lst@DefOther\lst@arg{#1}\let#2\@empty
```

```

1121 \expandafter\lst@DoAlso@\expandafter#2\expandafter#3\lst@arg\relax}
1122 \def\lst@DoAlso@#1#2#3{%
1123 \ifx\relax#3\expandafter\@gobblethree \else
... while not reaching \relax we use the TEXnique from \lst@SaveOutputDef
to replace the class by #2. Eventually we append the new definition to #1.
1124 \begingroup \lccode'\~='#3\relax \lowercase{\endgroup
1125 \def\lst@temp##1\def~##2##3\relax{%
1126 \edef\lst@arg{\def\noexpand~{\noexpand#2\expandafter
1127 \noexpand\@gobble##2}}}%
1128 \expandafter\lst@temp\lst@SelectStdCharTable\relax
1129 \lst@lExtend#1{\lst@arg}%
1130 \fi
1131 \lst@DoAlso@#1#2}

```

\lst@SaveDef These macros can be used in language definitions to make special changes. They
\lst@DefSaveDef save the definition and define or assign a new one.

```

\lst@LetSaveDef 1132 \def\lst@SaveDef#1#2{%
1133 \begingroup \lccode'\~='#1\relax \lowercase{\endgroup\let#2~}}
1134 \def\lst@DefSaveDef#1#2{%
1135 \begingroup \lccode'\~='#1\relax \lowercase{\endgroup\let#2~\def~}}
1136 \def\lst@LetSaveDef#1#2{%
1137 \begingroup \lccode'\~='#1\relax \lowercase{\endgroup\let#2~\let~}}

```

Now we get to the more powerful definitions.

\lst@CDef Here we unfold the first parameter $\langle 1st \rangle \{ \langle 2nd \rangle \} \{ \langle rest \rangle \}$ and say that this input string is ‘replaced’ by $\langle save\ 1st \rangle \{ \langle 2nd \rangle \} \{ \langle rest \rangle \}$ —plus $\langle execute \rangle$, $\langle pre \rangle$, and $\langle post \rangle$. This main work is done by **\lst@CDefIt**.

```

1138 \def\lst@CDef#1{\lst@CDef@#1}
1139 \def\lst@CDef@#1#2#3#4{\lst@CDefIt#1{#2}{#3}{#4#2#3}#4}

```

\lst@CDefX drops the input string.

```

1140 \def\lst@CDefX#1{\lst@CDefX@#1}
1141 \def\lst@CDefX@#1#2#3{\lst@CDefIt#1{#2}{#3}{}}

```

\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!

```

1142 \def\lst@CDefIt#1#2#3#4#5#6#7#8{%
1143 \ifx\@empty#2\@empty

```

For a single character we just execute the arguments in the correct order. You might want to go back to section 11.2 to look them up.

```

1144 \def#1{#6\def\lst@next{#7#4#8}\lst@next}%
1145 \else \ifx\@empty#3\@empty

```

For a two character sequence we test whether $\langle pre \rangle$ and $\langle post \rangle$ must be executed.

```

1146 \def#1##1{%
1147 #6%
1148 \ifx##1#2\def\lst@next{#7#4#8}\else
1149 \def\lst@next{#5##1}\fi
1150 \lst@next}%
1151 \else

```

We do the same for an arbitrary character sequence—except that we have to use `\lst@ifnextcharsarg` instead of `\ifx... \fi`.

```

1152      \def#1{%
1153          #6%
1154          \lst@ifnextcharsarg{#2#3}{#7#4#8}%
1155                                  {\expandafter#5\lst@eaten}}%
1156      \fi \fi}

```

`\lst@CArgX` We make `#1#2` active and call `\lst@CArg`.

```

1157 \def\lst@CArgX#1#2\relax{%
1158     \lst@DefActive\lst@arg{#1#2}%
1159     \expandafter\lst@CArg\lst@arg\relax}

```

`\lst@CArg` arranges the first two arguments for `\lst@CDef[X]`. We get an undefined macro and use `\@empty\@empty\relax` as delimiter for the submacro.

```

1160 \def\lst@CArg#1#2\relax{%
1161     \lccode'\='#1\lowercase{\def\lst@temp{}}%
1162     \lst@GetFreeMacro\lst@c\lst@temp}%
1163     \expandafter\lst@CArg@\lst@freemacro#1#2\@empty\@empty\relax}

```

Save meaning of $\langle 1st \rangle = \#2$ in $\langle save\ 1st \rangle = \#1$ and call the macro `#6` with correct arguments. From version 1.0 on, `#2`, `#3` and `#4` (respectively empty arguments) are tied together with group braces. This allows us to save two arguments in other definitions, for example in `\lst@DefDelimB`.

```

1164 \def\lst@CArg@#1#2#3#4\@empty#5\relax#6{%
1165     \let#1#2%
1166     \ifx\@empty#3\@empty
1167         \def\lst@next{#6{#2}{}}}%
1168     \else
1169         \def\lst@next{#6{#2#3{#4}}}%
1170     \fi
1171     \lst@next #1}

```

`\lst@CArgEmpty` ‘executes’ an `\@empty`-delimited argument. We will use it for the delimiters.

```

1172 \def\lst@CArgEmpty#1\@empty{#1}

```

15.4 Delimiters

Here we start with general definitions common to all delimiters.

`excludedelims` controls which delimiters are not printed in $\langle whatever \rangle$ style. We just define `\lst@ifex $\langle whatever \rangle$` to be true. Such switches are set false in the `ExcludeDelims` hook and are handled by the individual delimiters.

```

1173 \lst@Key{excludedelims}\relax
1174     {\lsthk@ExcludeDelims \lst@NormedDef\lst@temp{#1}%
1175     \expandafter\lst@for\lst@temp\do
1176     {\expandafter\let\csname\lst @ifex##1\endcsname\iftrue}}

```

`\lst@DelimPrint` And this macro might help in doing so. `#1` is `\lst@ifex $\langle whatever \rangle$` (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.

```

1177 \def\lst@DelimPrint#1#2{%

```

```

1178     #1%
1179     \begingroup
1180     \lst@mode\lst@nomode \lst@modetrue
1181     #2\lst@XPrintToken
1182     \endgroup
1183     \lst@ResetToken
1184     \fi}

```

`\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.

```

1185 \def\lst@DelimOpen#1#2#3#4#5#6\@empty{%
1186     \lst@TrackNewLines \lst@XPrintToken
1187     \lst@DelimPrint#1{#6}%
1188     \lst@EnterMode{#4}{\def\lst@currstyle#5}%
1189     \lst@DelimPrint{#1#2}{#6}%
1190     #3}

```

`\lst@DelimClose` is the same in reverse order.

```

1191 \def\lst@DelimClose#1#2#3\@empty{%
1192     \lst@TrackNewLines \lst@XPrintToken
1193     \lst@DelimPrint{#1#2}{#3}%
1194     \lst@LeaveMode
1195     \lst@DelimPrint{#1}{#3}}

```

`\lst@BeginDelim` These definitions are applications of `\lst@DelimOpen` and `\lst@DelimClose`: the `\lst@EndDelim` delimiters have the same style as the delimited text.

```

1196 \def\lst@BeginDelim{\lst@DelimOpen\iffalse\else{}}
1197 \def\lst@EndDelim{\lst@DelimClose\iffalse\else}

```

`\lst@BeginIDelim` Another application: no delimiter is printed.

```

\lst@EndIDelim 1198 \def\lst@BeginIDelim{\lst@DelimOpen\iffalse{}}
1199 \def\lst@EndIDelim{\lst@DelimClose\iffalse{}}

```

`\lst@DefDelims` This macro defines all delimiters and is therefore reset every language selection.

```

1200 \lst@AddToHook{SelectCharTable}{\lst@DefDelims}
1201 \lst@AddToHookExe{SetLanguage}{\let\lst@DefDelims\@empty}

```

`\lst@Delim` First we set default values: no `\lst@modetrue`, cumulative style, and no argument to `\lst@Delim[DM]@<type>`.

```

1202 \def\lst@Delim#1{%
1203     \lst@false \let\lst@cumulative\@empty \let\lst@arg\@empty

```

These are the correct settings for the double-star-form, so we immediately call the submacro in this case. Otherwise we either just suppress cumulative style, or even indicate the usage of `\lst@modetrue` with `\lst@true`.

```

1204     \@ifstar{\@ifstar{\lst@Delim@{#1}}%
1205                 {\let\lst@cumulative\relax
1206                 \lst@Delim@{#1}}}%
1207     {\lst@true\lst@Delim@{#1}}

```

The type argument is saved for later use. We check against the optional `<style>` argument using #1 as default, define `\lst@delimstyle` and look for the optional `<type option>`, which is just saved in `\lst@arg`.

```

1208 \def\lst@Delim@#1[#2]{%
1209     \gdef\lst@delimtype{#2}%
1210     \@ifnextchar[\lst@Delim@sty
1211         {\lst@Delim@sty[#1]}}
1212 \def\lst@Delim@sty[#1]{%
1213     \def\lst@delimstyle{#1}%
1214     \ifx\@empty#1\@empty\else
1215         \lst@Delim@sty@ #1\@nil
1216     \fi
1217     \@ifnextchar[\lst@Delim@option
1218         \lst@Delim@delim}
1219 \def\lst@Delim@option[#1]{\def\lst@arg{[#1]}\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`.

```

1220 \def\lst@Delim@sty@#1#2\@nil{%
1221     \if\relax\noexpand#1\else
1222         \edef\lst@delimstyle{\expandafter\noexpand
1223             \csname\@lst @\lst@delimstyle\endcsname}%
1224     \fi}

```

`\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.

```

1225 \def\lst@Delim@delim#1\relax#2#3#4#5#6#7#8{%
1226     \ifx #4\@empty \lst@Delim@delall{#2}\fi
1227     \ifx\@empty#1\@empty
1228         \ifx #4\@nil
1229             \@ifundefined{\@lst @#2DM@\lst@delimtype}%
1230                 {\lst@Delim@delall{#2@\lst@delimtype}}%
1231                 {\lst@Delim@delall{#2DM@\lst@delimtype}}%
1232         \fi
1233     \else

```

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.

```

1234     \expandafter\lst@Delim@args\expandafter
1235     {\lst@delimtype}{#1}{#5}{#6}{#7}{#8}{#4}%

```

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`.

```

1236     \let\lst@delim\@empty
1237     \expandafter\lst@ifOneOf\lst@delimtype\relax#3%
1238     {\@ifundefined{\@lst @#2DM@\lst@delimtype}%
1239         {\lst@lExtend\lst@delim{\csname\@lst @#2@\lst@delimtype
1240             \expandafter\endcsname\lst@arg}}%
1241         {\lst@lExtend\lst@delim{\expandafter\lst@UseDynamicMode
1242             \csname\@lst @#2DM@\lst@delimtype
1243             \expandafter\endcsname\lst@arg}}%

```

Now, depending on the mode #4 we either remove this particular delimiter or append it to all current ones.

```

1244     \ifx #4\@nil
1245         \let\lst@temp\lst@DefDelims \let\lst@DefDelims\@empty

```



```

1246         \expandafter\lst@Delim@del\lst@temp\@empty\@nil\@nil\@nil
1247     \else
1248         \lst@lExtend\lst@DefDelims\lst@delim
1249     \fi}%

    An unknown type issues an error.
1250     {\PackageError{Listings}{Illegal type ‘\lst@delimtype’}%
1251         {#2 types are #3.}}%
1252     \fi}

```

`\lst@Delim@args` Now let’s look how we add the arguments to `\lst@arg`. First we initialize the conversion just to make all characters active. But if the first character of the type equals #4, ...

```

1253 \def\lst@Delim@args#1#2#3#4#5#6#7{%
1254     \begingroup
1255     \lst@false \let\lst@next\lst@XConvert
    ... we remove that character from \lst@delimtype, and #5 might select a different
    conversion setting or macro.
1256     \@ifnextchar #4{\xdef\lst@delimtype{\expandafter\@gobble
1257         \lst@delimtype}%
1258         #5\lst@next#2\@nil
1259         \lst@lAddTo\lst@arg{\@empty#6}%
1260         \lst@GobbleNil}%

```

Since we are in the ‘special’ case above, we’ve also added the special `\lst@Begin...` and `\lst@End...` macros to `\lst@arg` (and `\@empty` as a brake for the delimiter). No special task must be done if the characters are not equal.

```

1261         {\lst@next#2\@nil
1262         \lst@lAddTo\lst@arg{\@empty#3}%
1263         \lst@GobbleNil}%
1264     #1\@nil

```

We always transfer the arguments to the outside of the group and append the style and mode selection if and only if we’re not deleting a delimiter. Therefor we expand the delimiter style.

```

1265     \global\let\@gtempa\lst@arg
1266     \endgroup
1267     \let\lst@arg\@gtempa
1268     \ifx #7\@nil\else
1269         \expandafter\lst@Delim@args@\expandafter{\lst@delimstyle}%
1270     \fi}

```

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@Delim[DM]@<type>`.

```

1271 \def\lst@Delim@args#1{%
1272     \lst@if
1273         \lst@lAddTo\lst@arg{{#1}\lst@modetrue}}%
1274     \else
1275         \ifx\lst@cumulative\@empty

```

```

1276         \lst@lAddTo\lst@arg{{{#1}}}%
1277     \else
1278         \lst@lAddTo\lst@arg{{{#1}}}%
1279     \fi
1280 \fi}

```

\lst@Delim@del To delete a particular delimiter, we iterate down the list of delimiters and compare the current item with the user supplied.

```

1281 \def\lst@Delim@del#1\@empty#2#3#4{%
1282     \ifx #2\@nil\else
1283         \def\lst@temp{#1\@empty#2#3}%
1284         \ifx\lst@temp\lst@delim\else
1285             \lst@lAddTo\lst@DefDelims{#1\@empty#2#3{#4}}%
1286         \fi
1287         \expandafter\lst@Delim@del
1288     \fi}

```

\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.

```

1289 \def\lst@Delim@delall#1{%
1290     \begingroup
1291     \edef\lst@delim{\expandafter\string\csname\lst @#1\endcsname}%
1292     \lst@false \global\let\@gtempa\@empty
1293     \expandafter\lst@Delim@delall@\lst@DefDelims\@empty
1294     \endgroup
1295     \let\lst@DefDelims\@gtempa}

```

We first discard a preceding `\lst@UseDynamicMode`.

```

1296 \def\lst@Delim@delall@#1{%
1297     \ifx #1\@empty\else
1298         \ifx #1\lst@UseDynamicMode
1299             \lst@true
1300             \let\lst@next\lst@Delim@delall@do
1301         \else
1302             \def\lst@next{\lst@Delim@delall@do#1}%
1303         \fi
1304         \expandafter\lst@next
1305     \fi}

```

Then we can check whether (the following) `\lst@<delimiter name>...` matches the delimiter class given by `\lst@delim`.

```

1306 \def\lst@Delim@delall@do#1#2\@empty#3#4#5{%
1307     \expandafter\lst@ifSubstring\expandafter{\lst@delim}{\string#1}%
1308     {%
1309         {\lst@if \lst@AddTo\@gtempa\lst@UseDynamicMode \fi
1310         \lst@AddTo\@gtempa{#1#2\@empty#3#4{#5}}}%
1311     \lst@false \lst@Delim@delall@}

```

\lst@DefDelimB 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`.

```

1312 \gdef\lst@DefDelimB#1#2#3#4#5#6#7#8{%
1313     \lst@CDef{#1}#2%
1314     {#3}%
1315     {\let\lst@bnext\lst@CArgEmpty

```

```

1316     \lst@ifmode #4\else
1317         #5%
1318         \def\lst@bnext{#6{#7}{#8}}%
1319     \fi
1320     \lst@bnext}%
1321 \@empty}

```

After a bug report from Vespe Savikko I added braces around #7.

`\lst@DefDelimE` The `\ifnum #7=\lst@mode` in the 5th line ensures that the delimiters match each other.

```

1322 \gdef\lst@DefDelimE#1#2#3#4#5#6#7{%
1323     \lst@CDef{#1}#2%
1324     {#3}%
1325     {\let\lst@enext\lst@CArgEmpty
1326     \ifnum #7=\lst@mode%
1327         #4%
1328         \let\lst@enext#6%
1329     \else
1330         #5%
1331     \fi
1332     \lst@enext}%
1333 \@empty}
1334 \lst@AddToHook{Init}{\let\lst@bnext\relax \let\lst@enext\relax}

```

`\lst@DefDelimBE` This service macro will actually define all string delimiters.

```

1335 \gdef\lst@DefDelimBE#1#2#3#4#5#6#7#8#9{%
1336     \lst@CDef{#1}#2%
1337     {#3}%
1338     {\let\lst@bnext\lst@CArgEmpty
1339     \ifnum #7=\lst@mode
1340         #4%
1341         \let\lst@bnext#9%
1342     \else
1343         \lst@ifmode\else
1344             #5%
1345             \def\lst@bnext{#6{#7}{#8}}%
1346         \fi
1347     \fi
1348     \lst@bnext}%
1349 \@empty}

```

`\lst@delimtypes` is the list of general delimiter types.

```

1350 \gdef\lst@delimtypes{s,l}

```

`\lst@DelimKey` We just put together the arguments for `\lst@Delim`.

```

1351 \gdef\lst@DelimKey#1#2{%
1352     \lst@Delim{#2}\relax
1353     {Delim}\lst@delimtypes #1%
1354     {\lst@BeginDelim\lst@EndDelim}
1355     i\@empty{\lst@BeginIDelim\lst@EndIDelim}}

```

`delim` all use `\lst@DelimKey`.

`moredelim`

`deletedelim`

```

1356 \lst@Key{delim}\relax{\lst@DelimKey\@empty{#1}}
1357 \lst@Key{moredelim}\relax{\lst@DelimKey\relax{#1}}
1358 \lst@Key{deletedelim}\relax{\lst@DelimKey\@nil{#1}}

```

`\lst@DelimDM@l` Nohting special here.

```

\lst@DelimDM@s 1359 \gdef\lst@DelimDM@l#1#2\@empty#3#4#5{%
1360     \lst@CArg #2\relax\lst@DefDelimB{#1}{#3}{#4}{#5\lst@Lmodetrue}}
1361 \gdef\lst@DelimDM@s#1#2#3\@empty#4#5#6{%
1362     \lst@CArg #2\relax\lst@DefDelimB{#1}{#3}{#4}{#5}{#6}%
1363     \lst@CArg #3\relax\lst@DefDelimE{#1}{#4}{#5}{#6}}
1364 \kernel

```

15.4.1 Strings

Just starting a new aspect.

```

1365 (*misc)
1366 \lst@BeginAspect{strings}

```

`\lst@stringtypes` is the list of ... string types?

```

1367 \gdef\lst@stringtypes{d,b,m,bd,db}

```

`\lst@StringKey` We just put together the arguments for `\lst@Delim`.

```

1368 \gdef\lst@StringKey#1#2{%
1369     \lst@Delim\lst@stringstyle #2\relax
1370     {String}\lst@stringtypes #1%
1371     {\lst@BeginString\lst@EndString}%
1372     \@end\@empty}}

```

`string` all use `\lst@StringKey`.

```

morestring 1373 \lst@Key{string}\relax{\lst@StringKey\@empty{#1}}
deletestring 1374 \lst@Key{morestring}\relax{\lst@StringKey\relax{#1}}
1375 \lst@Key{deletestring}\relax{\lst@StringKey\@nil{#1}}

```

`stringstyle` You shouldn't need comments on the following two lines, do you?

```

1376 \lst@Key{stringstyle}{#1}{\def\lst@stringstyle{#1}}
1377 \lst@AddToHook{EmptyStyle}{\let\lst@stringstyle\@empty}

```

`showstringspaces` Thanks to Knut Müller for reporting problems with `\blankstringtrue` (now `showstringspaces=false`). The problem has gone.

```

1378 \lst@Key{showstringspaces}t[t]{\lstKV@SetIf{#1}\lst@ifshowstringspaces}

```

`\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`.

```

1379 \gdef\lst@BeginString{%
1380     \lst@DelimOpen
1381     \lst@ifexstrings\else
1382     {\lst@ifshowstringspaces
1383         \lst@keepspacetrue
1384         \let\lst@outputspace\lst@visiblepace
1385     \fi}}

```

```
1386 \lst@AddToHookExe{ExcludeDelims}{\let\lst@ifexstrings\iffalse}
```

\lst@EndString Again the two tokens following \lst@DelimClose are arguments.

```
1387 \gdef\lst@EndString{\lst@DelimClose\lst@ifexstrings\else}
```

And now all the \lst@StringDM@*<type>* definitions.

\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.

```
1388 \gdef\lst@StringDM@d#1#2\@empty#3#4#5{%
```

```
1389     \lst@CArg #2\relax\lst@DefDelimBE{}{}#3{#1}{#5}#4}
```

\lst@StringDM@b The \lst@ifletter...\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.

```
1390 \gdef\lst@StringDM@b#1#2\@empty#3#4#5{%
```

```
1391     \let\lst@ifbstring\iftrue
```

```
1392     \lst@CArg #2\relax\lst@DefDelimBE
```

```
1393         {\lst@ifletter \lst@Output \lst@letterfalse \fi}%
```

```
1394         {\ifx\lst@lastother\lstum@backslash
```

```
1395             \expandafter\@gobblethree
```

```
1396             \fi}{}#3{#1}{#5}#4}
```

```
1397 \global\let\lst@ifbstring\iffalse % init
```

Heiko Heil reported problems with double backslashes. So:

```
1398 \lst@AddToHook{SelectCharTable}{%
```

```
1399     \lst@ifbstring
```

```
1400         \lst@CArgX \\\relax \lst@CDefX{}%
```

```
1401         {\lst@ProcessOther\lstum@backslash
```

```
1402         \lst@ProcessOther\lstum@backslash
```

```
1403         \let\lst@lastother\relax}%
```

```
1404         {}%
```

```
1405     \fi}
```

The reset of \lst@lastother has been added after a bug reports by Hermann Hüttler and Dan Luecking.

\lst@StringDM@bd are just the same and the same as \lst@StringDM@b.

```
\lst@StringDM@db 1406 \global\let\lst@StringDM@bd\lst@StringDM@b
```

```
1407 \global\let\lst@StringDM@db\lst@StringDM@bd
```

\lst@StringDM@a This delimiter type is designed for Ada. Here we enter string mode only if the last character has not been a letter and has not been a right parenthesis or right bracket. The test for the latter one has been added after bug report from Christian Kindinger.

```
1408 \gdef\lst@StringDM@a#1#2\@empty#3#4#5{%
```

```
1409     \lst@CArg #2\relax\lst@DefDelimBE{}{}%
```

```
1410         {\let\lst@next\@gobblethree
```

```
1411         \lst@ifletter\else
```

```
1412         \ifx\lst@lastother)\else \ifx\lst@lastother]\else
```

```
1413         \let\lst@next\@empty
```

```
1414         \fi \fi \fi
```

```
1415         \lst@next}#3{#1}{#5}#4}
```

`\lst@StringDM@m` 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.

```

1416 \gdef\lst@StringDM@m#1#2\@empty#3#4#5{%
1417     \lst@CArg #2\relax\lst@DefDelimBE{}{}%
1418     {\let\lst@next\@gobblethree
1419     \lst@ifletter\else
1420         \lst@ifLastOtherOneOf{)}].0123456789\lstum@rbrace'}%
1421     }%
1422     {\let\lst@next\@empty}%
1423     \fi
1424     \lst@next}#3{#1}{#5}#4}

```

`\lstum@rbrace` This has been used above.

```

1425 \lst@SaveOutputDef{"7D}\lstum@rbrace

1426 \lst@EndAspect
1427 </misc>

```

15.4.2 Comments

That's what we are working on.

```

1428 <*misc>
1429 \lst@BeginAspect{comments}

```

`\lst@commentmode` is a general purpose mode for comments.

```

1430 \lst@NewMode\lst@commentmode

```

`\lst@commenttypes` Via `comment` available comment types: line, fixed column, single, and nested and all with preceding `i` for invisible comments.

```

1431 \gdef\lst@commenttypes{l,f,s,n}

```

`\lst@CommentKey` We just put together the arguments for `\lst@Delim`.

```

1432 \gdef\lst@CommentKey#1#2{%
1433     \lst@Delim\lst@commentstyle #2\relax
1434     {Comment}\lst@commenttypes #1%
1435     {\lst@BeginComment\lst@EndComment}%
1436     i\@empty{\lst@BeginInvisible\lst@EndInvisible}}

```

`comment` The keys are easy since defined in terms of `\lst@CommentKey`.

`morecomment` 1437 `\lst@Key{comment}\relax{\lst@CommentKey\@empty{#1}}`

`deletecomment` 1438 `\lst@Key{morecomment}\relax{\lst@CommentKey\relax{#1}}`

1439 `\lst@Key{deletecomment}\relax{\lst@CommentKey\@nil{#1}}`

`commentstyle` Any hints necessary?

```

1440 \lst@Key{commentstyle}{}{\def\lst@commentstyle{#1}}
1441 \lst@AddToHook{EmptyStyle}{\let\lst@commentstyle\itshape}

```

`\lst@BeginComment` Once more the three tokens following `\lst@DelimOpen` are arguments.

```

\lst@EndComment 1442 \gdef\lst@BeginComment{%
1443     \lst@DelimOpen
1444     \lst@ifexcomments\else
1445     \lsthk@AfterBeginComment}

```

Ditto.

```
1446 \gdef\lst@EndComment{\lst@DelimClose\lst@ifexcomments\else}
1447 \lst@AddToHook{AfterBeginComment}{%}
1448 \lst@AddToHookExe{ExcludeDelims}{\let\lst@ifexcomments\iffalse}
```

`\lst@BeginInvisible` Print preceding characters and begin dropping the output.

```
\lst@EndInvisible 1449 \gdef\lst@BeginInvisible#1#2#3\@empty{%
1450   \lst@TrackNewLines \lst@XPrintToken
1451   \lst@BeginDropOutput{#1}}
```

Don't print the delimiter and end dropping the output.

```
1452 \gdef\lst@EndInvisible#1\@empty{\lst@EndDropOutput}
```

Now we provide all `\lst@Comment[DM]@⟨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.

```
1453 \gdef\lst@CommentDM@l#1#2\@empty#3#4#5{%
1454   \lst@CArg #2\relax\lst@DefDelimB{}{}#3{#1}{#5\lst@Lmodetrue}}
```

`\lst@CommentDM@f` is slightly more work. First we provide the number of preceding columns.

```
1455 \gdef\lst@CommentDM@f#1{%
1456   \@ifnextchar[{\lst@Comment@f{#1}}%
1457   {\lst@Comment@f{#1}[0]}}
```

We define the comment in the same way as above, but we enter comment mode if and only if the character is in column #2 (counting from zero).

```
1458 \gdef\lst@Comment@f#1[#2]#3\@empty#4#5#6{%
1459   \lst@CArg #3\relax\lst@DefDelimB{}{}%
1460   {\lst@CalcColumn
1461    \ifnum #2=\@tempcnta\else
1462     \expandafter\@gobblethree
1463     \fi}%
1464   #4{#1}{#6\lst@Lmodetrue}}
```

`\lst@CommentDM@s` Nothing special here.

```
1465 \gdef\lst@CommentDM@s#1#2#3\@empty#4#5#6{%
1466   \lst@CArg #2\relax\lst@DefDelimB{}{}#4{#1}{#6}%
1467   \lst@CArg #3\relax\lst@DefDelimE{}{}#5{#1}}
```

`\lst@CommentDM@n` We either give an error message or define the nested comment.

```
1468 \gdef\lst@CommentDM@n#1#2#3\@empty#4#5#6{%
1469   \ifx\@empty#3\@empty\else
1470     \def\@tempa{#2}\def\@tempb{#3}%
1471     \ifx\@tempa\@tempb
1472       \PackageError{Listings}{Identical delimiters}%
1473       {These delimiters make no sense with nested comments.}%
1474     \else
1475       \lst@CArg #2\relax\lst@DefDelimB
1476       {}%
```

Note that the following `\@gobble` eats an `\else` from `\lst@DefDelimB`.

```

1477             {\ifnum\lst@mode=#1\relax \expandafter\@gobble \fi}%
1478             {}#4{#1}{#6}%
1479             \lst@CArg #3\relax\lst@DefDelimE{ }{ }#5{#1}%
1480         \fi
1481     \fi}

1482 \lst@EndAspect
1483 </misc>

```

15.4.3 PODs

PODs are defined as a separate aspect.

```

1484 <*misc>
1485 \lst@BeginAspect{pod}

```

`printpod` We begin with the user keys, which I introduced after communication with Michael Piotrowski.

```

1486 \lst@Key{printpod}{false}[t]{\lstKV@SetIf{#1}\lst@ifprintpod}
1487 \lst@Key{podcomment}{false}[t]{\lstKV@SetIf{#1}\lst@ifpodcomment}
1488 \lst@AddToHookExe{SetLanguage}{\let\lst@ifpodcomment\iffalse}

```

`\lst@PODmode` is the static mode for PODs.

```

1489 \lst@NewMode\lst@PODmode

```

We adjust some characters if the user has selected `podcomment=true`.

```

1490 \lst@AddToHook{SelectCharTable}
1491     {\lst@ifpodcomment
1492         \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 printed. Moreover we gobble it if the current column number is not zero—`\@tempcnta` is valued below.

```

1493         {\ifnum\@tempcnta=\z@
1494             \lst@ifprintpod\else
1495                 \def\lst@bnext{\lst@BeginDropOutput\lst@PODmode}%
1496                 \expandafter\expandafter\expandafter\@gobblethree
1497             \fi
1498         \else
1499             \expandafter\@gobblethree
1500         \fi}%
1501         \lst@BeginComment\lst@PODmode{{\lst@commentstyle}}%

```

If we come to `=`, we calculate the current column number (zero based).

```

1502         \lst@CArgX =cut\^M\relax\lst@DefDelimE
1503         {\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'.

```

1504         {\ifnum\@tempcnta=\z@\else
1505             \expandafter\@gobblethree
1506         \fi}%

```



```

1507         {}%
1508         \lst@EndComment\lst@PODmode
1509     \fi}
1510 \lst@EndAspect
1511 </misc>

```

15.4.4 Tags

Support for HTML and other ‘markup languages’.

```

1512 <*misc>
1513 \lst@BeginAspect[keywords]{html}

```

`\lst@tagtypes` Again we begin with the list of tag types. It’s rather short.

```

1514 \gdef\lst@tagtypes{s}

```

`\lst@TagKey` Again we just put together the arguments for `\lst@Delim` and ...

```

1515 \gdef\lst@TagKey#1#2{%
1516     \lst@Delim\lst@tagstyle #2\relax
1517     {Tag}\lst@tagtypes #1%
1518         {\lst@BeginTag\lst@EndTag}%
1519     \@end\@empty{}}

```

`tag` ... we use the definition here.

```

1520 \lst@Key{tag}\relax{\lst@TagKey\@empty{#1}}

```

`tagstyle` You shouldn’t need comments on the following two lines, do you?

```

1521 \lst@Key{tagstyle}{}{\def\lst@tagstyle{#1}}
1522 \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.

```

1523 \gdef\lst@BeginTag{%
1524     \lst@DelimOpen
1525     \lst@ifextags\else
1526     {\let\lst@ifkeywords\iftrue
1527     \lst@ifmarkfirstintag \lst@firstintagtrue \fi}}
1528 \lst@AddToHookExe{ExcludeDelims}{\let\lst@ifextags\iffalse}

```

`\lst@EndTag` is just like the other `\lst@End<whatever>` definitions.

```

1529 \gdef\lst@EndTag{\lst@DelimClose\lst@ifextags\else}

```

`usekeywordsintag` The second key has already been ‘used’.

```

markfirstintag 1530 \lst@Key{usekeywordsintag}t[t]{\lstKV@SetIf{#1}\lst@ifusekeysintag}
1531 \lst@Key{markfirstintag}f[t]{\lstKV@SetIf{#1}\lst@ifmarkfirstintag}

```

For this, we install a (global) switch, ...

```

1532 \gdef\lst@firstintagtrue{\global\let\lst@iffirstintag\iftrue}
1533 \global\let\lst@iffirstintag\iffalse

```

... which is reset by the output of an identifier but not by other output.

```

1534 \lst@AddToHook{PostOutput}{\lst@tagresetfirst}
1535 \lst@AddToHook{Output}
1536     {\gdef\lst@tagresetfirst{\global\let\lst@iffirstintag\iffalse}}
1537 \lst@AddToHook{OutputOther}{\gdef\lst@tagresetfirst{}}

```

Now we only need to test against this switch in the `Output` hook.

```
1538 \lst@AddToHook{Output}
1539     {\ifnum\lst@mode=\lst@tagmode
1540         \lst@iffirstintag \let\lst@thestyle\lst@gkeywords@sty \fi
1541         \lst@ifusekeysintag\else \let\lst@thestyle\lst@gkeywords@sty\fi
1542     \fi}
```

Moreover we check here, whether the keyword style is always to be used.

`\lst@tagmode` We allocate the mode and ...

```
1543 \lst@NewMode\lst@tagmode
1544 \lst@AddToHook{Init}{\global\let\lst@ifnotag\iftrue}
1545 \lst@AddToHook{SelectCharTable}{\let\lst@ifkeywords\lst@ifnotag}
```

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.

```
1546 \gdef\lst@Tag@s#1#2\@empty#3#4#5{%
1547     \global\let\lst@ifnotag\iffalse
1548     \lst@CArg #1\relax\lst@DefDelimB {}{}%
1549     {\ifnum\lst@mode=\lst@tagmode \expandafter\@gobblethree \fi}%
1550     #3\lst@tagmode{#5}%
1551     \lst@CArg #2\relax\lst@DefDelimE {}{}{}#4\lst@tagmode}%
```

`\lst@BeginCDATA` This macro is used by the XML language definition.

```
1552 \gdef\lst@BeginCDATA#1\@empty{%
1553     \lst@TrackNewLines \lst@PrintToken
1554     \lst@EnterMode\lst@GPmode{}\let\lst@ifmode\iffalse
1555     \lst@mode\lst@tagmode #1\lst@mode\lst@GPmode\relax\lst@modetrue}

1556 \lst@EndAspect
1557 </misc>
```

15.5 Replacing input

1558 `<*kernel>`

`\lst@ReplaceInput` is defined in terms of `\lst@CArgX` and `\lst@CDefX`.

```
1559 \def\lst@ReplaceInput#1{\lst@CArgX #1\relax\lst@CDefX{}{}}
```

`literate` Jason Alexander asked for something like that. The key simply saves the argument.

```
1560 \lst@Key{literate}{}{\def\lst@literate{#1}}
1561 \lst@AddToHook{SelectCharTable}
1562     {\ifx\lst@literate\@empty\else
1563         \expandafter\lst@Literate\lst@literate{}\relax\z@
1564     \fi}
```

Internally we make use of the 'replace input' feature. We print the preceding text, assign token and length, and output it.

```
1565 \def\lst@Literate#1#2#3{%
1566     \ifx\relax#2\@empty\else
1567         \lst@ReplaceInput{#1}%
```

```

1568         {\ifx\lst@OutputBox\@gobble\else
1569           \lst@XPrintToken \let\lst@scanmode\lst@scan@m
1570           \lst@token{#2}\lst@length#3\relax
1571           \lst@XPrintToken
1572         \fi}%
1573     \expandafter\lst@Literate
1574 \fi}

```

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.

```

1575 \def\lst@BeginDropInput#1{%
1576   \lst@EnterMode{#1}%
1577   {\lst@modetrue
1578     \let\lst@OutputBox\@gobble
1579     \let\lst@ifdropinput\iftrue
1580     \let\lst@ProcessLetter\@gobble
1581     \let\lst@ProcessDigit\@gobble
1582     \let\lst@ProcessOther\@gobble
1583     \let\lst@ProcessSpace\@empty
1584     \let\lst@ProcessTabulator\@empty
1585     \let\lst@ProcessFormFeed\@empty}}
1586 \let\lst@ifdropinput\iffalse % init
1587 </kernel>

```

15.6 Escaping to L^AT_EX

We now define the ... damned ... the aspect has escaped!

```

1588 < *misc>
1589 \lst@BeginAspect{escape}

```

`texcl` Communication with Jörn Wilms is responsible for this key. The definition and the first hooks are easy.

```

1590 \lst@Key{texcl}{false}[t]{\lstKV@SetIf{#1}\lst@iftexcl}
1591 \lst@AddToHook{TextStyle}{\let\lst@iftexcl\iffalse}
1592 \lst@AddToHook{EOL}
1593   {\ifnum\lst@mode=\lst@TeXLmode
1594     \expandafter\lst@escapeend
1595     \expandafter\lst@LeaveAllModes
1596     \expandafter\lst@ReenterModes
1597   \fi}

```

If the user wants T_EX comment lines, we print the comment separator and interrupt the normal processing.

```

1598 \lst@AddToHook{AfterBeginComment}
1599   {\lst@iftexcl \lst@ifLmode \lst@ifdropinput\else
1600     \lst@PrintToken
1601     \lst@LeaveMode \lst@InterruptModes
1602     \lst@EnterMode{\lst@TeXLmode}{\lst@modetrue\lst@commentstyle}%
1603     \expandafter\expandafter\expandafter\lst@escapebegin
1604   \fi \fi \fi}

```

```
1605 \lst@NewMode\lst@TeXLmode
```

\lst@ActiveCDefX Same as **\lst@CDefX** but we both make #1 active and assign a new catcode.

```
1606 \gdef\lst@ActiveCDefX#1{\lst@ActiveCDefX@#1}
1607 \gdef\lst@ActiveCDefX@#1#2#3{
1608   \catcode'#1\active\lccode'\~='#1%
1609   \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.

```
1610 \gdef\lst@Escape#1#2#3#4{%
1611   \lst@CArgX #1\relax\lst@CDefX
1612   {}%
1613   {\lst@ifdropinput\else
1614     \lst@TrackNewLines\lst@OutputLostSpace \lst@XPrintToken
1615     \lst@InterruptModes
1616     \lst@EnterMode{\lst@TeXmode}{\lst@modetrue}%
```

Now we must define the character sequence to end the escape.

```
1617   \ifx\~M#2%
1618     \lst@CArg #2\relax\lst@ActiveCDefX
1619     {}%
1620     {\lst@escapeend #4\lst@LeaveAllModes\lst@ReenterModes}%
1621     {\lst@MProcessListing}%
1622   \else
1623     \lst@CArg #2\relax\lst@ActiveCDefX
1624     {}%
1625     {\lst@escapeend #4\lst@LeaveAllModes\lst@ReenterModes
1626       \lst@whitespacefalse}%
1627     {}%
1628   \fi
1629   #3\lst@escapebegin
1630   \fi}%
1631   {}}}
```

The **\lst@whitespacefalse** above was added after a bug report from Martin Steffen.

```
1632 \lst@NewMode\lst@TeXmode
```

escapebegin The keys simply store the arguments.

```
escapeend 1633 \lst@Key{escapebegin}{-}{\def\lst@escapebegin{#1}}
1634 \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.

```
1635 \lst@Key{escapechar}{-}
1636   {\ifx\@empty#1\@empty
1637     \let\lst@DefEsc\relax
1638   \else
1639     \def\lst@DefEsc{\lst@Escape{#1}{#1}{-}{-}}%
1640   \fi}
1641 \lst@AddToHook{TextStyle}{\let\lst@DefEsc\@empty}
1642 \lst@AddToHook{SelectCharTable}{\lst@DefEsc}
```

`escapeinside` Nearly the same.

```

1643 \lst@Key{escapeinside}{-}{\lstKV@TwoArg{#1}%
1644     {\let\lst@DefEsc\@empty
1645     \ifx\@empty##1@empty\else \ifx\@empty##2\@empty\else
1646         \def\lst@DefEsc{\lst@Escape{##1}{##2}{-}}}%
1647     \fi\fi}}

```

`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.

```

1648 \lst@Key{mathescape}{false}[t]{\lstKV@SetIf{#1}\lst@ifmathescape}
1649 \lst@AddToHook{SelectCharTable}
1650     {\lst@ifmathescape \lst@Escape{\$}{\$}%
1651     {\setbox\@tempboxa=\hbox\bgroup$}%
1652     {$\egroup \lst@CalcLostSpaceAndOutput}\fi}

1653 \lst@EndAspect
1654 </misc>

```

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.

```

1655 <*misc>
1656 \lst@BeginAspect{keywords}

1657 \global\let\lst@ifensitive\iftrue % init
1658 \global\let\lst@ifensitivedefed\iffalse % init % \global

```

All keyword tests take the following three arguments.

```

#1 = <prefix>
#2 = \lst@<name>@list (a list of macros which contain the keywords)
#3 = \lst@g<name>@sty (global style macro)

```

We begin with non memory-saving tests.

```

1659 \lst@ifsavemem\else

```

`\lst@KeywordTest` 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`.

```

1660 \gdef\lst@KeywordTest#1#2#3{%
1661     \begingroup \let\lst@UM\@empty
1662     \global\expandafter\let\expandafter\@gtempa
1663         \csname\@lst#1@the\lst@token\endcsname
1664     \endgroup
1665     \ifx\@gtempa\relax\else
1666         \let\lst@thestyle\@gtempa
1667     \fi}

```

Note that we need neither #2 nor #3 here.

`\lst@KEYWORDTEST` Case insensitive tests make the current character string upper case and give it to a submacro similar to `\lst@KeywordTest`.

```

1668 \gdef\lst@KEYWORDTEST{%
1669     \uppercase\expandafter{\expandafter
1670         \lst@KEYWORDTEST@the\lst@token}\relax}
1671 \gdef\lst@KEYWORDTEST@#1\relax#2#3#4{%
1672     \begingroup \let\lst@UM\@empty
1673     \global\expandafter\let\expandafter\@gtempa
1674         \csname\@lst#2@#1\endcsname
1675     \endgroup
1676     \ifx\@gtempa\relax\else
1677         \let\lst@thestyle\@gtempa
1678     \fi}

```

`\lst@WorkingTest` The same except that `\lst<prefix>@<current token>` might be a working procedure;
`\lst@WORKINGTEST` it is executed.

```

1679 \gdef\lst@WorkingTest#1#2#3{%
1680     \begingroup \let\lst@UM\@empty
1681     \global\expandafter\let\expandafter\@gtempa
1682         \csname\@lst#1@the\lst@token\endcsname
1683     \endgroup
1684     \@gtempa}

1685 \gdef\lst@WORKINGTEST{%
1686     \uppercase\expandafter{\expandafter
1687         \lst@WORKINGTEST@the\lst@token}\relax}
1688 \gdef\lst@WORKINGTEST@#1\relax#2#3#4{%
1689     \begingroup \let\lst@UM\@empty
1690     \global\expandafter\let\expandafter\@gtempa
1691         \csname\@lst#2@#1\endcsname
1692     \endgroup
1693     \@gtempa}

```

`\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, ...

```

1694 \gdef\lst@DefineKeywords#1#2#3{%
1695     \lst@ifensitive
1696         \def\lst@next{\lst@for#2}%
1697     \else
1698         \def\lst@next{\uppercase\expandafter{\expandafter\lst@for#2}}%
1699     \fi
1700     \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.

```

1701     {\expandafter\ifx\csname\@lst#1@##1\endcsname\relax
1702         \global\expandafter\let\csname\@lst#1@##1\endcsname#3%
1703     \fi}}

```

`\lst@UndefineKeywords` We make the keywords upper case if necessary, ...

```

1704 \gdef\lst@UndefineKeywords#1#2#3{%
1705     \lst@ifsensitivedefed
1706         \def\lst@next{\lst@for#2}%
1707     \else
1708         \def\lst@next{\uppercase\expandafter{\expandafter\lst@for#2}}%
1709     \fi
1710     \lst@next\do
    ... iterate through the list, and ‘undefine’ \lst<prefix>@<keyword> if it’s equivalent
    to \lst@g<name>@sty.
1711     {\expandafter\ifx\csname\@lst#1@##1\endcsname#3%
1712         \global\expandafter\let\csname\@lst#1@##1\endcsname\relax
1713     \fi}}
```

Thanks to Magnus Lewis-Smith a wrong #2 in the replacement text could be changed to #3.

And now memory-saving tests.

```

1714 \fi
1715 \lst@ifsavemem
```

`\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.

```

1716 \gdef\lst@ifOneOutOf#1\relax#2{%
1717     \def\lst@temp##1,#1,##2##3\relax{%
1718         \ifx\@empty##2\else \expandafter\lst@I000first \fi}%
1719     \def\lst@next{\lst@ifOneOutOf@#1\relax}%
1720     \expandafter\lst@next#2\relax\relax}
```

We either execute the *<else>* part or make the next test.

```

1721 \gdef\lst@ifOneOutOf@#1\relax#2#3{%
1722     \ifx#2\relax
1723         \expandafter\@secondoftwo
1724     \else
1725         \expandafter\lst@temp\expandafter,#2,#1,\@empty\relax
1726         \expandafter\lst@next
1727     \fi}
1728 \ifx\iffalse\else\fi
1729 \gdef\lst@I000first#1\relax#2#3{\fi#2}
```

The line `\ifx\iffalse\else\fi` balances the `\fi` inside `\lst@I000first`.

`\lst@ifONEOUTOF` As in `\lst@ifONEOF` we need two `\uppercase`s here.

```

1730 \gdef\lst@ifONEOUTOF#1\relax#2{%
1731     \uppercase{\def\lst@temp##1,#1,##2##3\relax}%
1732     \ifx\@empty##2\else \expandafter\lst@I000first \fi}%
1733     \def\lst@next{\lst@ifONEOUTOF@#1\relax}%
1734     \expandafter\lst@next#2\relax}
1735 \gdef\lst@ifONEOUTOF@#1\relax#2#3{%
1736     \ifx#2\relax
1737         \expandafter\@secondoftwo
1738     \else
1739         \uppercase
1740         {\expandafter\lst@temp\expandafter,#2,#1,\@empty\relax}%
```

```

1741      \expandafter\lst@next
1742    \fi}

```

Note: The third last line uses the fact that keyword lists (not the list of keyword lists) are already made upper case if keywords are insensitive.

`\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.

```

1743 \gdef\lst@KWTest{%
1744   \begingroup \let\lst@UM\@empty
1745   \expandafter\xdef\expandafter\@gtempa\expandafter{\the\lst@token}%
1746   \endgroup
1747   \expandafter\lst@ifOneOutOf\@gtempa\relax}

```

`\lst@KeywordTest` are fairly easy now. Note that we don't need `#1=<prefix>` here.

```

\lst@KEYWORDTEST 1748 \gdef\lst@KeywordTest#1#2#3{\lst@KWTest #2{\let\lst@thestyle#3}{}}
1749 \global\let\lst@KEYWORDTEST\lst@KeywordTest

```

For case insensitive tests we assign the insensitive version to `\lst@ifOneOutOf`. Thus we need no extra definition here.

`\lst@WorkingTest` Ditto.

```

\lst@WORKINGTEST 1750 \gdef\lst@WorkingTest#1#2#3{\lst@KWTest #2#3{}}
1751 \global\let\lst@WORKINGTEST\lst@WorkingTest

1752 \fi

```

`sensitive` is a switch, preset true every language selection.

```

1753 \lst@Key{sensitive}\relax[t]{\lstKV@SetIf{#1}\lst@ifensitive}
1754 \lst@AddToHook{SetLanguage}{\let\lst@ifensitive\iftrue}

```

We select case insensitive definitions if necessary.

```

1755 \lst@AddToHook{Init}
1756   {\lst@ifensitive\else
1757     \let\lst@KeywordTest\lst@KEYWORDTEST
1758     \let\lst@WorkingTest\lst@WORKINGTEST
1759     \let\lst@ifOneOutOf\lst@IFONEOUTOF
1760   \fi}

```

`\lst@MakeMacroUppercase` makes the contents of `#1` (if defined) upper case.

```

1761 \gdef\lst@MakeMacroUppercase#1{%
1762   \ifx\@undefined#1\else \uppercase\expandafter
1763     {\expandafter\def\expandafter#1\expandafter{#1}}%
1764   \fi}

```

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.

```

1765 \gdef\lst@InstallTest#1#2#3#4#5#6#7#8{%
1766     \lst@AddToHook{TrackKeywords}{\lst@TrackKeywords{#1}#2#4#6#7#8}%
1767     \lst@AddToHook{PostTrackKeywords}{\lst@PostTrackKeywords#2#3#4#5}}

1768 \lst@AddToHook{Init}{\lsthk@TrackKeywords\lsthk@PostTrackKeywords}
1769 \lst@AddToHook{TrackKeywords}{}% init
1770 \lst@AddToHook{PostTrackKeywords}{}% init

```

We have to detect the keywords somewhere.

```

1771 \lst@AddToHook{Output}{\lst@ifkeywords \lsthk@DetectKeywords \fi}
1772 \lst@AddToHook{DetectKeywords}{}% init
1773 \lst@AddToHook{ModeTrue}{\let\lst@ifkeywords\iffalse}
1774 \lst@AddToHook{Init}{\let\lst@ifkeywords\iftrue}

```

`\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@<name>@list \lst@g<name>@sty` to the DetectKeywords hook.

```

1775 \gdef\lst@InstallTestNow#1#2#3#4#5{%
1776     \@ifundefined{string#2#1}%
1777     {\global\@namedef{string#2#1}}}%
1778     \edef\@tempa{%
1779         \noexpand\lst@AddToHook{\ifx#5dDetectKeywords\else Output\fi}%
1780         {\ifx #4w\noexpand\lst@WorkingTest
1781             \else\noexpand\lst@KeywordTest \fi
1782             {#1}\noexpand#2\noexpand#3}}%

```

If we are advised to save memory, we insert a test for each `<name>`. Otherwise we install the tests according to `<prefix>`.

```

1783     \lst@ifsavemem
1784     \@tempa
1785     \else
1786         \@ifundefined{\@lst#1@if@ins}%
1787         {\@tempa \global\@namedef{\@lst#1@if@ins}}}%
1788         {}%
1789     \fi}
1790     {}

```

`\lst@TrackKeywords` Now it gets a bit tricky. We expand the class list `\lst@<name>@list` behind `\lst@TK@{<prefix>}\lst@g<name>@sty` and use two `\relaxes` as terminators. This will define the keywords of all the classes as keywords of type `<prefix>`. More details come soon.

```

1791 \gdef\lst@TrackKeywords#1#2#3#4#5#6{%
1792     \lst@false
1793     \def\lst@arg{{#1}#4}%

```

```

1794 \expandafter\expandafter\expandafter\lst@TK@
1795 \expandafter\lst@arg#2\relax\relax

```

And nearly the same to undefine all out-dated keywords, which is necessary only if we don't save memory.

```

1796 \lst@ifsavemem\else
1797 \def\lst@arg#{#1}#4#2}%
1798 \expandafter\expandafter\expandafter\lst@TK@@
1799 \expandafter\lst@arg#3\relax\relax
1800 \fi

```

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.

```

1801 \lst@if \lst@InstallTestNow{#1}#2#4#5#6\fi}

```

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@` are

```

#1 = <prefix>
#2 = \lst@g<name>@sty
#3 = \lst@<id>
#4 = \lst@g<id>

```

```

1802 \gdef\lst@TK@#1#2#3#4{%
1803 \ifx\lst@ifensitive\lst@ifensitivedefed
1804 \ifx#3#4\else
1805 \lst@true
1806 \lst@ifsavemem\else
1807 \lst@UndefineKeywords{#1}#4#2%
1808 \lst@DefineKeywords{#1}#3#2%
1809 \fi
1810 \fi
1811 \else
1812 \ifx#3\relax\else
1813 \lst@true
1814 \lst@ifsavemem\else
1815 \lst@UndefineKeywords{#1}#4#2%
1816 \lst@DefineKeywords{#1}#3#2%
1817 \fi
1818 \fi
1819 \fi

```

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.

```

1820 \lst@ifsavemem \ifx#3\relax\else
1821 \lst@ifensitive\else \lst@MakeMacroUppercase#3\fi
1822 \fi \fi

```

Reaching the end of the class list, we end the loop.

```

1823 \ifx#3\relax
1824 \expandafter\@gobblethree
1825 \fi
1826 \lst@TK@{#1}#2}

```

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 = <prefix>
#2 = \lst@g<name>@sty
#3 = \lst@<name>@list
#4 = \lst@<id>
#5 = \lst@g<id>

1827 \gdef\lst@TK@@#1#2#3#4#5{%
1828     \ifx#4\relax
1829         \expandafter\@gobblefour
1830     \else
1831         \lst@ifSubstring{#4#5}#3{-}\lst@UndefineKeywords{#1}#5#2}%
1832     \fi
1833     \lst@TK@@{#1}#2#3}

```

Keywords are up-to-date after InitVars.

```

1834 \lst@AddToHook{InitVars}
1835     {\global\let\lst@ifsensitivedefed\lst@ifensitive}

```

`\lst@PostTrackKeywords` After updating all the keywords, the global keywords and the global list become equivalent to the local ones.

```

1836 \gdef\lst@PostTrackKeywords#1#2#3#4{%
1837     \lst@ifsavemem\else
1838         \global\let#3#1%
1839         \global\let#4#2%
1840     \fi}

```

16.3 Classes and families

`classoffset` just stores the argument in a macro.

```

1841 \lst@Key{classoffset}\z@{\def\lst@classoffset{#1}}

```

`\lst@InstallFamily` Recall the parameters

```

#1 = <prefix>
#2 = <name>
#3 = <style name>
#4 = <style init>
#5 = <default style name>
#6 = <working procedure>
#7 = l|o (language or other key)
#8 = d|o (DetectKeywords or Output hook)

```

First we define the keys and the style key `<style name>` if and only if the name is not empty.

```

1842 \gdef\lst@InstallFamily#1#2#3#4#5{%
1843     \lst@Key{#2}\relax{\lst@UseFamily{#2}##1\relax\lst@MakeKeywords}%
1844     \lst@Key{more#2}\relax
1845         {\lst@UseFamily{#2}##1\relax\lst@MakeMoreKeywords}%
1846     \lst@Key{delete#2}\relax
1847         {\lst@UseFamily{#2}##1\relax\lst@DeleteKeywords}%
1848     \ifx\@empty#3\@empty\else

```

```

1849 \lst@Key{#3}{#4}{\lstKV@OptArg[\@one]{##1}%
1850 {\@tempcnta\lst@classoffset \advance\@tempcnta###1\relax
1851 \namedef{lst@#3\ifnum\@tempcnta=\@one\else \the\@tempcnta
1852 \fi}{###2}}}%
1853 \fi
1854 \expandafter\lst@InstallFamily@
1855 \csname\@lst @#2\data\expandafter\endcsname
1856 \csname\@lst @#5\endcsname {#1}{#2}{#3}}

```

Now we check whether *working procedure* is empty. Accordingly we use working procedure or style in the ‘data’ definition. The working procedure is defined right here if necessary.

```

1857 \gdef\lst@InstallFamily@#1#2#3#4#5#6#7#8{%
1858 \gdef#1{{#3}{#4}{#5}{#2#7}}%
1859 \long\def\lst@temp##1{#6}%
1860 \ifx\lst@temp\gobble
1861 \lst@AddTo#1{s#8}%
1862 \else
1863 \lst@AddTo#1{w#8}%
1864 \global\@namedef{lst@g#4@wp}##1{#6}%
1865 \fi}

```

Nothing else is defined here, all the rest is done on demand.

`\lst@UseFamily` We look for the optional class number, provide this member, ...

```

1866 \gdef\lst@UseFamily#1{%
1867 \def\lst@family{#1}%
1868 \@ifnextchar[\lst@UseFamily@{\lst@UseFamily@[\@one]}}
1869 \gdef\lst@UseFamily@#1{%
1870 \@tempcnta\lst@classoffset \advance\@tempcnta#1\relax
1871 \lst@ProvideFamily\lst@family
... and build the control sequences ...
1872 \lst@UseFamily@a
1873 {\lst@family\ifnum\@tempcnta=\@one\else \the\@tempcnta \fi}}
1874 \gdef\lst@UseFamily@a#1{%
1875 \expandafter\lst@UseFamily@b
1876 \csname\@lst @#1@list\expandafter\endcsname
1877 \csname\@lst @#1\expandafter\endcsname
1878 \csname\@lst @#1@also\expandafter\endcsname
1879 \csname\@lst @g#1\endcsname}

```

... required for `\lst@MakeKeywords` and #6.

```

1880 \gdef\lst@UseFamily@b#1#2#3#4#5\relax#6{\lstKV@XOptArg[] {#5}{#6#1#2#3#4}}

```

`\lst@ProvideFamily` provides the member ‘`\the\@tempcnta`’ of the family #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 `lstfam` instead of `lst`. The marker `\lstfam@#1[number]` is defined globally since a bug report by Edsko de Vries.

```

1881 \gdef\lst@ProvideFamily#1{%
1882 \@ifundefined{lstfam@#1\ifnum\@tempcnta=\@one\else\the\@tempcnta\fi}%
1883 {\global\@namedef{lstfam@#1\ifnum\@tempcnta=\@one\else
1884 \the\@tempcnta\fi}}}%
1885 \expandafter\expandafter\expandafter\lst@ProvideFamily@

```

```

1886      \csname\@lst @#1\data\endcsname
1887      {\ifnum\@tempcnta=\@ne\else \the\@tempcnta \fi}}%
1888  {}}%

```

Now we have the following arguments

```

#1 = <prefix>
#2 = <name>
#3 = <style name>
#4 = <default style name>
#5 = l|o (language or other key)
#6 = w|s (working procedure or style)
#7 = d|o (DetectKeywords or Output hook)
#8 = \ifnum\@tempcnta=\@ne\else \the\@tempcnta \fi

```

We define `\lst@g<name><number>@sty` to call either `\lst@g<name>@wp` with the number as argument or `\lst@<style name><number>` where the number belongs to the control sequence.

```

1889 \gdef\lst@ProvideFamily@#1#2#3#4#5#6#7#8{%
1890   \expandafter\xdef\csname\@lst @g#2#8@sty\endcsname
1891   {\if #6w%
1892     \expandafter\noexpand\csname\@lst @g#2@wp\endcsname{#8}%
1893   \else
1894     \expandafter\noexpand\csname\@lst @#3#8\endcsname
1895   \fi}%

```

We ensure the existence of the style macro. This is done in the `Init` hook by assigning the default style if necessary.

```

1896   \ifx\@empty#3\@empty\else
1897     \edef\lst@temp{\noexpand\lst@AddToHook{Init}}{%
1898       \noexpand\lst@ProvideStyle\expandafter\noexpand
1899       \csname\@lst @#3#8\endcsname\noexpand#4}}%
1900   \lst@temp
1901   \fi

```

We call a submacro to do the rest. It requires some control sequences.

```

1902   \expandafter\lst@ProvideFamily@@
1903   \csname\@lst @#2#8@list\expandafter\endcsname
1904   \csname\@lst @#2#8\expandafter\endcsname
1905   \csname\@lst @#2#8@also\expandafter\endcsname
1906   \csname\@lst @g#2#8@list\expandafter\endcsname
1907   \csname\@lst @g#2#8\expandafter\endcsname
1908   \csname\@lst @g#2#8@sty\expandafter\endcsname
1909   {#1}#5#6#7}

```

Now we have (except that `<number>` is possibly always missing)

```

#1 = \lst@<name><number>@list
#2 = \lst@<name><number>
#3 = \lst@<name><number>@also
#4 = \lst@g<name><number>@list
#5 = \lst@g<name><number>
#6 = \lst@g<name><number>@sty
#7 = <prefix>
#8 = l|o (language or other key)
#9 = w|s (working procedure or style)

```

#10 = d|o (DetectKeywords or Output hook)

Note that #9 and ‘#10’ are read by \lst@InstallTest. We initialize all required ‘variables’ (at SetLanguage) and install the test (which definition is in fact also delayed).

```

1910 \gdef\lst@ProvideFamily@@#1#2#3#4#5#6#7#8{%
1911     \gdef#1{#2#5}\global\let#2\empty \global\let#3\empty % init
1912     \gdef#4{#2#5}\global\let#5\empty % init
1913     \if #8\relax
1914         \lst@AddToHook{SetLanguage}{\def#1{#2#5}\let#2\empty}%
1915     \fi
1916     \lst@InstallTest{#7}#1#2#4#5#6}

```

\lst@InstallKeywords Now we take advance of the optional argument construction above. Thus, we just insert [\@ne] as *<number>* in the definitions of the keys.

```

1917 \gdef\lst@InstallKeywords#1#2#3#4#5{%
1918     \lst@Key{#2}\relax
1919     {\lst@UseFamily{#2}[\@ne]##1\relax\lst@MakeKeywords}%
1920     \lst@Key{more#2}\relax
1921     {\lst@UseFamily{#2}[\@ne]##1\relax\lst@MakeMoreKeywords}%
1922     \lst@Key{delete#2}\relax
1923     {\lst@UseFamily{#2}[\@ne]##1\relax\lst@DeleteKeywords}%
1924     \ifx\@empty#3\@empty\else
1925         \lst@Key{#3}{#4}{\@namedef\lst@#3{##1}}%
1926     \fi
1927     \expandafter\lst@InstallFamily@
1928         \csname\@lst @#2\data\expandafter\endcsname
1929         \csname\@lst @#5\endcsname {#1}{#2}{#3}}

```

\lst@ProvideStyle If the style macro #1 is not defined, it becomes equivalent to #2.

```

1930 \gdef\lst@ProvideStyle#1#2{%
1931     \ifx#1\@undefined \let#1#2%
1932     \else\ifx#1\relax \let#1#2\fi\fi}

```

Finally we define \lst@MakeKeywords, ..., \lst@DeleteKeywords. We begin with two helper.

\lst@BuildClassList After #1 follows a comma separated list of keyword classes terminated by ,\relax,, e.g. keywords2,emph1,\relax,. For each *<item>* in this list we *append* the two macros \lst@*<item>*\lst@g*<item>* to #1.

```

1933 \gdef\lst@BuildClassList#1#2,{%
1934     \ifx\relax#2\@empty\else
1935         \ifx\@empty#2\@empty\else
1936             \lst@lExtend#1{\csname\@lst @#2\expandafter\endcsname
1937                 \csname\@lst @g#2\endcsname}%
1938         \fi
1939     \expandafter\lst@BuildClassList\expandafter#1
1940     \fi}

```

\lst@DeleteClassesIn deletes pairs of tokens, namely the arguments #2#3 to the submacro.

```

1941 \gdef\lst@DeleteClassesIn#1#2{%
1942     \expandafter\lst@DCI@\expandafter#1#2\relax\relax}
1943 \gdef\lst@DCI@#1#2#3{%
1944     \ifx#2\relax

```

```

1945         \expandafter\@gobbletwo
1946     \else
        If we haven't reached the end of the class list, we define a temporary macro which
        removes all appearances.
1947         \def\lst@temp##1#2#3##2{%
1948             \lst@lAddTo#1{##1}%
1949             \ifx ##2\relax\else
1950                 \expandafter\lst@temp
1951             \fi ##2}%
1952         \let\@tempa#1\let#1\@empty
1953         \expandafter\lst@temp\@tempa#2#3\relax
1954     \fi
1955     \lst@DCI@#1}

```

\lst@MakeKeywords We empty some macros and make use of **\lst@MakeMoreKeywords**. Note that this and the next two definitions have the following arguments:

```

#1 = class list (in brackets)
#2 = keyword list
#3 = \lst@<name>@list
#4 = \lst@<name>
#5 = \lst@<name>@also
#6 = \lst@g<name>

```

```

1956 \gdef\lst@MakeKeywords[#1]#2#3#4#5#6{%
1957     \def#3{#4#6}\let#4\@empty \let#5\@empty
1958     \lst@MakeMoreKeywords[#1]{#2}#3#4#5#6}

```

\lst@MakeMoreKeywords We append classes and keywords.

```

1959 \gdef\lst@MakeMoreKeywords[#1]#2#3#4#5#6{%
1960     \lst@BuildClassList#3#1,\relax,%
1961     \lst@DefOther\lst@temp{,#2}\lst@lExtend#4\lst@temp}

```

\lst@DeleteKeywords We convert the keyword arguments via **\lst@MakeKeywords** and remove the classes and keywords.

```

1962 \gdef\lst@DeleteKeywords[#1]#2#3#4#5#6{%
1963     \lst@MakeKeywords[#1]{#2}\@tempa\@tempb#5#6%
1964     \lst@DeleteClassesIn#3\@tempa
1965     \lst@DeleteKeysIn#4\@tempb}

```

16.4 Main families and classes

Keywords

keywords Defining the keyword family gets very, very easy.

```

1966 \lst@InstallFamily k{keywords}{keywordstyle}\bfseries{keywordstyle}{\ld

```

ndkeywords Second order keywords use the same trick as **\lst@InstallKeywords**.

```

1967 \lst@Key{ndkeywords}\relax
1968     {\lst@UseFamily{keywords}[\tw@]#1\relax\lst@MakeKeywords}%
1969 \lst@Key{morendkeywords}\relax
1970     {\lst@UseFamily{keywords}[\tw@]#1\relax\lst@MakeMoreKeywords}%
1971 \lst@Key{deletendkeywords}\relax
1972     {\lst@UseFamily{keywords}[\tw@]#1\relax\lst@DeleteKeywords}%
1973 \lst@Key{ndkeywordstyle}\relax{\@namedef{\lst@keywordstyle2}{#1}}%

```

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.

```

1974 \lst@Key{keywordsprefix}\relax{\lst@DefActive\lst@keywordsprefix{#1}}
1975 \global\let\lst@keywordsprefix\empty
1976 \lst@AddToHook{SelectCharTable}
1977     {\ifx\lst@keywordsprefix\empty\else
1978         \expandafter\lst@CArg\lst@keywordsprefix\relax
1979         \lst@CDef{}}%
1980         {\lst@ifletter\else
1981             \global\let\lst@prefixkeyword\empty
1982             \fi}%
1983         {}%
1984     \fi}
1985 \lst@AddToHook{Init}{\global\let\lst@prefixkeyword\relax}
1986 \lst@AddToHook{Output}
1987     {\ifx\lst@prefixkeyword\empty
1988         \let\lst@thestyle\lst@gkeywords@sty
1989         \global\let\lst@prefixkeyword\relax
1990     \fi}%

```

`otherkeywords` 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.

```

1991 \lst@Key{otherkeywords}{-}{%
1992     \let\lst@otherkeywords\empty
1993     \lst@for{#1}\do{%
1994         \lst@MakeActive{##1}%
1995         \lst@lExtend\lst@otherkeywords{%
1996             \expandafter\lst@CArg\lst@temp\relax\lst@CDef
1997             {} \lst@PrintOtherKeyword\empty}}}
1998 \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.

```

1999 \gdef\lst@PrintOtherKeyword#1\@empty{%
2000     \lst@XPrintToken
2001     \begingroup
2002         \lst@modetrue \lsthk@TextStyle
2003         \let\lst@ProcessDigit\lst@ProcessLetter
2004         \let\lst@ProcessOther\lst@ProcessLetter
2005         \lst@lettertrue
2006         \lst@gkeywords@sty{#1\lst@XPrintToken}%
2007     \endgroup}

```

To do: Which part of `TextStyle` hook is required?

```

2008 \lst@EndAspect
2009 \</misc>

```


The emphasize family

is just one macro call here.

```
2010 ⟨*misc⟩
2011 \lst@BeginAspect[keywords]{emph}
2012 \lst@InstallFamily e{emph}{emphstyle}{}{emphstyle}{}od
2013 \lst@EndAspect
2014 ⟨/misc⟩
```

T_PX control sequences

Here we check the last ‘other’ processed token.

```
2015 ⟨*misc⟩
2016 \lst@BeginAspect[keywords]{tex}
2017 \lst@InstallKeywords{cs}{texcs}{texcsstyle}\relax{keywordstyle}
2018     {\ifx\lst@lastother\lstum@backslash
2019         \let\lst@thestyle\lst@texcsstyle
2020         \fi}
2021     ld
2022 \lst@EndAspect
2023 ⟨/misc⟩
```

Compiler directives

First some usual stuff.

```
directives2024 ⟨*misc⟩
2025 \lst@BeginAspect[keywords]{directives}

The initialization of \lst@directives has been added after a bug report from
Kris Luyten.

2026 \lst@NewMode\lst@CDmode
2027 \lst@AddToHook{EOL}{\ifnum\lst@mode=\lst@CDmode \lst@LeaveMode \fi}
2028 \lst@InstallKeywords{d}{directives}{directivestyle}\relax{keywordstyle}
2029     {\ifnum\lst@mode=\lst@CDmode
2030         \let\lst@thestyle\lst@directivestyle
2031         \fi}
2032     ld
2033 \global\let\lst@directives\@empty % init

Now we define a new delimiter for directives: We enter ‘directive mode’ only in
the first column.

2034 \lst@AddTo\lst@delimtypes{,directive}
2035 \gdef\lst@Delim@directive#1\@empty#2#3#4{%
2036     \lst@CArg #1\relax\lst@DefDelimB
2037     {\lst@CalcColumn}%
2038     }%
2039     {\ifnum\@tempcnta=\z@
2040         \def\lst@bnext{#2\lst@CDmode{#4\lst@Lmodetrue}%
2041         \let\lst@currstyle\lst@directivestyle}%
2042     \fi
2043     \@gobblethree}%
2044     #2\lst@CDmode{#4\lst@Lmodetrue}}
```

We introduce a new string type (thanks to R. Isernhagen), which ...

```

2045 \lst@AddTo\lst@stringtypes{,directive}
2046 \gdef\lst@StringDM@directive#1#2#3\@empty{%
2047     \lst@CArg #2\relax\lst@CDef
2048     {}}%
... is active only in \lst@CDmode:
2049     {\let\lst@bnext\lst@CArgEmpty
2050     \ifnum\lst@mode=\lst@CDmode
2051         \def\lst@bnext{\lst@BeginString{#1}}%
2052     \fi
2053     \lst@bnext}%
2054     \@empty
2055 \lst@CArg #3\relax\lst@CDef
2056 {}%
2057 {\let\lst@enext\lst@CArgEmpty
2058 \ifnum #1=\lst@mode
2059     \let\lst@bnext\lst@EndString
2060 \fi
2061 \lst@bnext}%
2062 \@empty}
2063 \lst@EndAspect
2064 \misc

```

16.5 Keyword comments

includes both comment types and is possibly split into this and `dkcs`.

```

2065 (*misc)
2066 \lst@BeginAspect[keywords,comments]{keywordcomments}

```

`\lst@BeginKC` Starting a keyword comment is easy, but: (1) The submacros are called outside of
`\lst@BeginKCS` two group levels, and ...

```

2067 \lst@NewMode\lst@KCmode \lst@NewMode\lst@KCSmode
2068 \gdef\lst@BeginKC{\aftergroup\aftergroup\aftergroup\lst@BeginKC@}%
2069 \gdef\lst@BeginKC@{%
2070     \lst@ResetToken
2071     \lst@BeginComment\lst@KCmode{{\lst@commentstyle}\lst@modetrue}%
2072     \@empty}%
2073 \gdef\lst@BeginKCS{\aftergroup\aftergroup\aftergroup\lst@BeginKCS@}%
2074 \gdef\lst@BeginKCS@{%
2075     \lst@ResetToken
2076     \lst@BeginComment\lst@KCSmode{{\lst@commentstyle}\lst@modetrue}%
2077     \@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`.

```

2078 \lst@AddToHook{PostOutput}{\lst@KCpost \global\let\lst@KCpost\@empty}
2079 \global\let\lst@KCpost\@empty % init

```

`\lst@EndKC` leaves the comment mode before the (temporarily saved) comment delimiter is printed.

```

2080 \gdef\lst@EndKC{\lst@SaveToken \lst@LeaveMode \lst@RestoreToken
2081     \let\lst@thestyle\lst@identifierstyle \lsthk@Output}

```

keywordcomment The delimiters must be identical here, thus we use `\lst@KCmatch`. Note the last argument `o` to `\lst@InstallKeywords`: The working test is installed in the **Output** hook and not in **DetectKeywords**. Otherwise we couldn't detect the ending delimiter since keyword detection is done if and only if mode changes are allowed.

```

2082 \lst@InstallKeywords{kc}{keywordcomment}{}\relax{}
2083     {\ifnum\lst@mode=\lst@KCmode
2084         \edef\lst@temp{\the\lst@token}%
2085         \ifx\lst@temp\lst@KCmatch
2086             \lst@EndKC
2087         \fi
2088     \else
2089         \lst@ifmode\else
2090             \xdef\lst@KCmatch{\the\lst@token}%
2091             \global\let\lst@KCpost\lst@BeginKC
2092         \fi
2093     \fi}
2094     lo

```

keywordcommentsemicolon The key simply stores the keywords. After a bug report by Norbert Eisinger the initialization in **SetLanguage** has been added.

```

2095 \lst@Key{keywordcommentsemicolon}{}{\lstKV@ThreeArg{#1}%
2096     {\def\lst@KCAkeywordsB{##1}%
2097     \def\lst@KCAkeywordsE{##2}%
2098     \def\lst@KCBkeywordsB{##3}%
2099     \def\lst@KCkeywords{##1##2##3}}}
2100 \lst@AddToHook{SetLanguage}{%
2101     \let\lst@KCAkeywordsB\@empty \let\lst@KCAkeywordsE\@empty
2102     \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 `\@emptys`.

```

2103 \lst@AddToHook{SelectCharTable}
2104     {\ifx\lst@KCkeywords\@empty\else
2105         \lst@DefSaveDef{'\;'}\lst@EKC
2106         {\lst@XPrintToken
2107             \ifnum\lst@mode=\lst@KCmode \lst@EndComment\@empty \else
2108             \ifnum\lst@mode=\lst@KCSmode \lst@EndComment\@empty
2109             \fi \fi
2110         \lst@EKC}%
2111     \fi}

```

The 'working identifier' macros enter respectively leave comment mode.

```

2112 \gdef\lst@KCAWorkB{%
2113     \lst@ifmode\else \global\let\lst@KCpost\lst@BeginKC \fi}
2114 \gdef\lst@KCBWorkB{%
2115     \lst@ifmode\else \global\let\lst@KCpost\lst@BeginKCS \fi}
2116 \gdef\lst@KCAWorkE{\ifnum\lst@mode=\lst@KCmode \lst@EndKC \fi}

```

Now we install the tests and initialize the given macros.

```

2117 \lst@ProvideFamily@@
2118     \lst@KCAkeywordsB@list\lst@KCAkeywordsB \lst@KC@also
2119     \lst@gKCAkeywordsB@list\lst@gKCAkeywordsB \lst@KCAWorkB
2120     {kcb}owo % prefix, other key, working procedure, Output hook

```

```

2121 \lst@ProvideFamily@@
2122     \lst@KCAkeywordsE@list\lst@KCAkeywordsE \lst@KC@also
2123     \lst@gKCAkeywordsE@list\lst@gKCAkeywordsE \lst@KCAWorkE
2124     {kce}owo
2125 \lst@ProvideFamily@@
2126     \lst@KCBkeywordsB@list\lst@KCBkeywordsB \lst@KC@also
2127     \lst@gKCBkeywordsB@list\lst@gKCBkeywordsB \lst@KCBWorkB
2128     {kcs}owo

2129 \lst@EndAspect
2130 </misc>

```

16.6 Export of identifiers

One more ‘keyword’ class.

```

\lstindexmacro 2131 (*misc)
2132 \lst@BeginAspect[keywords]{index}
2133 \lst@InstallFamily w{index}{indexstyle}\lstindexmacro{indexstyle}
2134     {\csname\lst @indexstyle#1\expandafter\endcsname
2135         \expandafter{\the\lst@token}}
2136     od
2137 \lst@UserCommand\lstindexmacro#1{\index{{\ttfamily#1}}}
2138 \lst@EndAspect
2139 </misc>

```

The ‘idea’ here is the usage of a global `\lst@ifprocname`, indicating a preceding `procnamestyle` ‘procedure keyword’. All the other is known stuff.

```

procnamekeys 2140 (*misc)
indexprocnames 2141 \lst@BeginAspect[keywords]{procnames}
2142 \gdef\lst@procnametrue{\global\let\lst@ifprocname\iftrue}
2143 \gdef\lst@procnamefalse{\global\let\lst@ifprocname\iffalse}
2144 \lst@AddToHook{Init}{\lst@procnamefalse}
2145 \lst@AddToHook{DetectKeywords}
2146     {\lst@ifprocname
2147         \let\lst@thestyle\lst@procnamestyle
2148         \lst@ifindexproc \csname\lst @gindex@sty\endcsname \fi
2149         \lst@procnamefalse
2150     \fi}

2151 \lst@Key{procnamestyle}{\def\lst@procnamestyle{#1}}
2152 \lst@Key{indexprocnames}{false}[t]{\lstKV@SetIf{#1}\lst@ifindexproc}
2153 \lst@AddToHook{Init}{\lst@ifindexproc \lst@indexproc \fi}
2154 \gdef\lst@indexproc{%
2155     \@ifundefined{lst@indexstyle1}%
2156     {\@namedef{lst@indexstyle1}##1{}}%
2157     {}

```

The default definition of `\lst@indexstyle` above has been moved outside the hook after a bug report from Ulrich G. Wortmann.

```

2158 \lst@InstallKeywords w{procnamekeys}{\relax}
2159     {\global\let\lst@PNpost\lst@procnametrue}
2160     od
2161 \lst@AddToHook{PostOutput}{\lst@PNpost\global\let\lst@PNpost\@empty}
2162 \global\let\lst@PNpost\@empty % init

```

```

2163 \lst@EndAspect
2164 </misc>

```

17 More aspects and keys

basicstyle There is no better place to define these keys, I think.

inputencoding

```

2165 (*kernel)
2166 \lst@Key{basicstyle}\relax{\def\lst@basicstyle{#1}}
2167 \lst@Key{inputencoding}\relax{\def\lst@inputenc{#1}}
2168 \lst@AddToHook{Init}
2169     {\lst@basicstyle
2170      \ifx\lst@inputenc\@empty\else
2171        \@ifundefined{inputencoding}{}%
2172        {\inputencoding\lst@inputenc}%
2173      \fi}
2174 \lst@AddToHookExe{EmptyStyle}
2175     {\let\lst@basicstyle\@empty
2176      \let\lst@inputenc\@empty}
2177 </kernel>

```

Michael Niedermair asked for a key like `inputencoding`.

17.1 Styles and languages

We begin with style definition and selection.

```

2178 <*misc>
2179 \lst@BeginAspect{style}

```

\lststylefiles This macro is defined if and only if it's undefined yet.

```

2180 \@ifundefined{lststylefiles}
2181     {\lst@UserCommand\lststylefiles{lststy0.sty}}{}

```

\lstdefinestyle are defined in terms of `\lst@DefStyle`, which is defined via `\lst@DefDriver`.

```

\lst@definestyle 2182 \lst@UserCommand\lstdefinestyle{\lst@DefStyle\iftrue}
\lst@DefStyle 2183 \lst@UserCommand\lstdefinestyle{\lst@DefStyle\iffalse}
2184 \gdef\lst@DefStyle{\lst@DefDriver{style}{sty}\lstset}

```

The ‘empty’ style calls the initial empty hook `EmptyStyle`.

```

2185 \global\@namedef{lststy$}\lsthk@EmptyStyle}
2186 \lst@AddToHook{EmptyStyle}{}% init

```

style is an application of `\lst@LAS`. We just specify the hook and an empty argument as ‘pre’ and ‘post’ code.

```

2187 \lst@Key{style}\relax{%
2188     \lst@LAS{style}{sty}{[] {#1}}\lst@NoAlias\lststylefiles
2189     \lsthk@SetStyle
2190     {}}
2191 \lst@AddToHook{SetStyle}{}% init

```

```

2192 \lst@EndAspect
2193 </misc>

```

Now we deal with commands used in defining and selecting programming languages, in particular with aliases.

```
2194 <misc>
2195 \lst@BeginAspect{language}
```

`\lstlanguagefiles` This macro is defined if and only if it's undefined yet.

```
2196 \@ifundefined{lstdriverfiles}
2197   {\lst@UserCommand\lstlanguagefiles{lstlang0.sty}}{}
```

`\lstdefinelanguage` are defined in terms of `\lst@DefLang`, which is defined via `\lst@DefDriver`.

```
\lst@definelanguage 2198 \lst@UserCommand\lstdefinelanguage{\lst@DefLang\iftrue}
\lst@DefLang 2199 \lst@UserCommand\lstdefinelanguage{\lst@DefLang\iffalse}
2200 \gdef\lst@DefLang{\lst@DefDriver{language}{lang}\lstset}
```

Now we can provide the ‘empty’ language.

```
2201 \lstdefinelanguage{}{}
```

`language` is mainly an application of `\lst@LAS`.

```
alsolanguage 2202 \lst@Key{language}\relax{\lstKV@OptArg[] {#1}%
2203   {\lst@LAS{language}{lang}{##1}{##2}}\lst@FindAlias\lstlanguagefiles
2204   \lsthk@SetLanguage
2205   {\lst@FindAlias [##1]{##2}%
2206   \let\lst@language\lst@alias
2207   \let\lst@dialect\lst@oalias}}}
```

Ditto, we simply don't execute `\lsthk@SetLanguage`.

```
2208 \lst@Key{alsolanguage}\relax{\lstKV@OptArg[] {#1}%
2209   {\lst@LAS{language}{lang}{##1}{##2}}\lst@FindAlias\lstlanguagefiles
2210   {}%
2211   {\lst@FindAlias [##1]{##2}%
2212   \let\lst@language\lst@alias
2213   \let\lst@dialect\lst@oalias}}}
```

```
2214 \lst@AddToHook{SetLanguage}{}% init
```

`\lstalias` Now we concentrate on aliases and default dialects. `\lsta@<language>$<dialect>` and `\lsta@<language>` contain the aliases of a particular dialect respectively a complete language. We'll use a `$`-character to separate a language name from its dialect.

```
2215 \lst@UserCommand\lstalias{\@ifnextchar[\lstalias@\lstalias@@}
2216 \gdef\lstalias@[#1]#2[#3]#4{\lst@NormedNameDef{\lsta@#2$#1}{#4$#3}}
2217 \gdef\lstalias@@#1#2{\lst@NormedNameDef{\lsta@#1}{#2}}
```

`defaultdialect` We simply store the dialect.

```
2218 \lst@Key{defaultdialect}\relax
2219   {\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.

```
2220 \gdef\lst@FindAlias [ #1 ] #2 { %
2221   \lst@NormedDef\lst@oalias { #1 } %
2222   \lst@NormedDef\lst@alias { #2 } %
2223   \@ifundefined{\lsta@\lst@alias} { } %
2224   {\edef\lst@alias {\csname\lst a@\lst@alias\endcsname}} %
```

```

2225 \ifx\@empty\lst@oalias \ifundefined{lstdd@\lst@malias}{}%
2226 {\edef\lst@oalias{\csname\@lst dd@\lst@malias\endcsname}}%
2227 \fi

```

Now we are ready for an alias of a single dialect.

```

2228 \edef\lst@temp{\lst@malias $\lst@oalias}%
2229 \ifundefined{lsta@\lst@temp}{}%
2230 {\edef\lst@temp{\csname\@lst a@\lst@temp\endcsname}}%

```

Finally we again set the default dialect—for the case of a dialect alias.

```

2231 \expandafter\lst@FindAlias@\lst@temp $}
2232 \gdef\lst@FindAlias@#1$#2${%
2233 \def\lst@malias{#1}\def\lst@oalias{#2}%
2234 \ifx\@empty\lst@oalias \ifundefined{lstdd@\lst@malias}{}%
2235 {\edef\lst@oalias{\csname\@lst dd@\lst@malias\endcsname}}%
2236 \fi}

```

`\lst@RequireLanguages` This definition will be equivalent to `\lstloadlanguages`. We requested the given list of languages and load additionally required aspects.

```

2237 \gdef\lst@RequireLanguages#1{%
2238 \lst@Require{language}{lang}{#1}\lst@FindAlias\lstlanguagefiles
2239 \ifx\lst@loadaspects\@empty\else
2240 \lst@RequireAspects\lst@loadaspects
2241 \fi}

```

`\lstloadlanguages` is the same as `\lst@RequireLanguages`.

```

2242 \global\let\lstloadlanguages\lst@RequireLanguages
2243 \lst@EndAspect
2244 \</misc>

```

17.2 Format definitions*

```

2245 \<*misc>
2246 \lst@BeginAspect{formats}

```

`\lstformatfiles` This macro is defined if and only if it's undefined yet.

```

2247 \ifundefined{lstformatfiles}
2248 {\lst@UserCommand\lstformatfiles{lstfmt0.sty}}{}

```

`\lstdefineformat` are defined in terms of `\lst@DefFormat`, which is defined via `\lst@DefDriver`.

```

\lst@defineformat 2249 \lst@UserCommand\lstdefineformat{\lst@DefFormat\iftrue}
\lst@DefFormat 2250 \lst@UserCommand\lstdefineformat{\lst@DefFormat\iffalse}
2251 \gdef\lst@DefFormat{\lst@DefDriver{format}{fmt}\lst@UseFormat}

```

We provide the ‘empty’ format.

```

2252 \lstdefineformat{}{}

```

`format` is an application of `\lst@LAS`. We just specify the hook as ‘pre’ and an empty argument as ‘post’ code.

```

2253 \lst@Key{format}\relax{%
2254 \lst@LAS{format}{fmt}{[] {#1}}\lst@NoAlias\lstformatfiles
2255 \lsthk@SetFormat
2256 {}
2257 \lst@AddToHook{SetFormat}{\let\lst@fmtformat\@empty}% init

```

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.

```

2258 \gdef\lst@fmtSplit#1#2{%
2259     \def\lst@temp##1#2##2\relax##3{%
2260         \ifnum##3=\z@
2261             \ifx\@empty##2\@empty
2262                 \lst@false
2263                 \let\lst@fmta#1%
2264                 \let\lst@fmtb\@empty
2265             \else
2266                 \expandafter\lst@temp#1\relax\@ne
2267             \fi
2268         \else
2269             \def\lst@fmta{##1}\def\lst@fmtb{##2}%
2270             \fi}%
2271     \lst@true
2272     \expandafter\lst@temp#1#2\relax\z@}

```

`\lst@ifNextCharWhitespace` is defined in terms of `\lst@ifSubstring`.

```

2273 \gdef\lst@ifNextCharWhitespace#1#2#3{%
2274     \lst@ifSubstring#3\lst@whitespaces{#1}{#2}#3}

```

And here come all white space characters.

```

2275 \begingroup
2276 \catcode'\^^I=12\catcode'\^^J=12\catcode'\^^M=12\catcode'\^^L=12\relax%
2277 \lst@DefActive\lst@whitespaces{\^^I^^J^^M}% add ^^L
2278 \global\let\lst@whitespaces\lst@whitespaces%
2279 \endgroup

```

`\lst@fmtIfIdentifier` tests the first character of #1

```

2280 \gdef\lst@fmtIfIdentifier#1{%
2281     \ifx\relax#1\@empty
2282         \expandafter\@secondoftwo
2283     \else
2284         \expandafter\lst@fmtIfIdentifier@\expandafter#1%
2285     \fi}

```

against the 'letters' `_`, `@`, `A`, ..., `Z` and `a`, ..., `z`.

```

2286 \gdef\lst@fmtIfIdentifier@#1#2\relax{%
2287     \let\lst@next\@secondoftwo
2288     \ifnum'#1='_\else
2289         \ifnum'#1<64\else
2290             \ifnum'#1<91\let\lst@next\@firstoftwo\else
2291             \ifnum'#1<97\else
2292                 \ifnum'#1<123\let\lst@next\@firstoftwo\else
2293                 \fi \fi \fi \fi
2294             \lst@next}

```

`\lst@fmtIfNextCharIn` is required for the optional (*exceptional characters*). The implementation is easy—refer section 13.1.


```

2295 \gdef\lst@fmtIfNextCharIn#1{%
2296     \ifx\@empty#1\@empty \expandafter\@secondoftwo \else
2297         \def\lst@next{\lst@fmtIfNextCharIn@{#1}}%
2298         \expandafter\lst@next\fi}
2299 \gdef\lst@fmtIfNextCharIn@#1#2#3#4{%
2300     \def\lst@temp##1#4##2##3\relax{%
2301         \ifx \@empty##2\expandafter\@secondoftwo
2302             \else \expandafter\@firstoftwo \fi}%
2303     \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.

```

2304 \gdef\lst@fmtCDef#1{\lst@fmtCDef@#1}
2305 \gdef\lst@fmtCDef@#1#2#3#4#5#6#7{%
2306     \lst@CDefIt#1{#2}{#3}%
2307         {\lst@fmtIfNextCharIn{#5}{#4#2#3}{#6#4#2#3#7}}%
2308         #4%
2309         {}{}{}}

```

`\lst@fmtCDefX` The same but ‘drop input’.

```

2310 \gdef\lst@fmtCDefX#1{\lst@fmtCDefX@#1}
2311 \gdef\lst@fmtCDefX@#1#2#3#4#5#6#7{%
2312     \let#4#1%
2313     \ifx\@empty#2\@empty
2314         \def#1{\lst@fmtIfNextCharIn{#5}{#4}{#6#7}}%
2315     \else \ifx\@empty#3\@empty
2316         \def#1#1{%
2317             \ifx##1#2%
2318                 \def\lst@next{\lst@fmtIfNextCharIn{#5}{#4##1}}%
2319                 {#6#7}}%
2320             \else
2321                 \def\lst@next{#4##1}%
2322             \fi
2323             \lst@next}%
2324     \else
2325         \def#1{%
2326             \lst@ifNextCharsArg{#2#3}%
2327                 {\lst@fmtIfNextCharIn{#5}{\expandafter#4\lst@eaten}%
2328                 {#6#7}}%
2329                 {\expandafter#4\lst@eaten}}%
2330     \fi \fi}

```

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.

```

2331 \gdef\lst@UseFormat#1{%
2332     \def\lst@fmtwhole{#1}%
2333     \lst@UseFormat@}
2334 \gdef\lst@UseFormat@{%
2335     \lst@fmtSplit\lst@fmtwhole,%

```

We assign the rest of the format definition, ...

```

2336 \let\lst@fmtwhole\lst@fmtb
2337 \ifx\lst@fmta\@empty\else
... split the item at the equal sign, and work on the item.
2338 \lst@fmtSplit\lst@fmta=%
2339 \ifx\@empty\lst@fmta\else

To do: Insert \let\lst@arg\@empty \expandafter\lst@XConvert\lst@fmtb\@nil
\let\lst@fmtb\lst@arg.

2340 \expandafter\lstKV@XOptArg\expandafter[\expandafter]%
2341 \expandafter{\lst@fmtb}\lst@UseFormat@b
2342 \fi
2343 \fi

```

Finally we process the next item if the rest is not empty.

```

2344 \ifx\lst@fmtwhole\@empty\else
2345 \expandafter\lst@UseFormat@
2346 \fi}

```

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`.

```

2347 \gdef\lst@UseFormat@b[#1]#2{%
2348 \def\lst@fmtc{{#1}}\lst@lExtend\lst@fmtc{\expandafter{\lst@fmta}}%
2349 \def\lst@fmtb{#2}%
2350 \lst@fmtSplit\lst@fmtb\string

```

We append an empty argument or `\lst@fmtPre` with ‘`\string`-preceding’ tokens as argument. We do the same for the tokens after `\string`.

```

2351 \ifx\@empty\lst@fmta
2352 \lst@lAddTo\lst@fmtc{{}}%
2353 \else
2354 \lst@lExtend\lst@fmtc{\expandafter
2355 {\expandafter\lst@fmtPre\expandafter{\lst@fmta}}}%
2356 \fi
2357 \ifx\@empty\lst@fmtb
2358 \lst@lAddTo\lst@fmtc{{}}%
2359 \else
2360 \lst@lExtend\lst@fmtc{\expandafter
2361 {\expandafter\lst@fmtPost\expandafter{\lst@fmtb}}}%
2362 \fi

```

Eventually we extend `\lst@fmtformat` appropriately. Note that `\lst@if` still indicates whether the replacement tokens contain `\string`.

```

2363 \expandafter\lst@UseFormat@c\lst@fmtc}
2364 \gdef\lst@UseFormat@c#1#2#3#4{%
2365 \lst@fmtIfIdentifier#2\relax
2366 {\lst@fmtIdentifier{#2}%
2367 \lst@if\else \PackageWarning{Listings}%
2368 {Cannot drop identifier in format definition}%
2369 \fi}%
2370 {\lst@if
2371 \lst@lAddTo\lst@fmtformat{\lst@CArgX#2\relax\lst@fmtCDef}%

```

```

2372 \else
2373 \lst@lAddTo\lst@fmtformat{\lst@CArgX#2\relax\lst@fmtCDefX}%
2374 \fi
2375 \lst@DefActive\lst@fmtc{#1}%
2376 \lst@lExtend\lst@fmtformat{\expandafter{\lst@fmtc}{#3}{#4}}}}
2377 \lst@AddToHook{SelectCharTable}{\lst@fmtformat}
2378 \global\let\lst@fmtformat\@empty

```

The formatting

\lst@fmtPre

```

2379 \gdef\lst@fmtPre#1{%
2380 \lst@PrintToken
2381 \begingroup
2382 \let\newline\lst@fmtEnsureNewLine
2383 \let\space\lst@fmtEnsureSpace
2384 \let\indent\lst@fmtIndent
2385 \let\noindent\lst@fmtNoindent
2386 #1%
2387 \endgroup}

```

\lst@fmtPost

```

2388 \gdef\lst@fmtPost#1{%
2389 \global\let\lst@fmtPostOutput\@empty
2390 \begingroup
2391 \def\newline{\lst@AddTo\lst@fmtPostOutput\lst@fmtEnsureNewLine}%
2392 \def\space{\aftergroup\lst@fmtEnsurePostSpace}%
2393 \def\indent{\lst@AddTo\lst@fmtPostOutput\lst@fmtIndent}%
2394 \def\noindent{\lst@AddTo\lst@fmtPostOutput\lst@fmtNoindent}%
2395 \aftergroup\lst@PrintToken
2396 #1%
2397 \endgroup}
2398 \lst@AddToHook{Init}{\global\let\lst@fmtPostOutput\@empty}
2399 \lst@AddToHook{PostOutput}
2400 {\lst@fmtPostOutput \global\let\lst@fmtPostOutput\@empty}

```

\lst@fmtEnsureSpace

```

\lst@fmtEnsurePostSpace 2401 \gdef\lst@fmtEnsureSpace{%
2402 \lst@ifwhitespace\else \expandafter\lst@ProcessSpace \fi}
2403 \gdef\lst@fmtEnsurePostSpace{%
2404 \lst@ifNextCharWhitespace{}{\lst@ProcessSpace}}

```

fmtindent

```

\lst@fmtIndent 2405 \lst@Key{fmtindent}{20pt}{\def\lst@fmtindent{#1}}
\lst@fmtNoindent 2406 \newdimen\lst@fmtcurrindent
2407 \lst@AddToHook{InitVars}{\global\lst@fmtcurrindent\z@}
2408 \gdef\lst@fmtIndent{\global\advance\lst@fmtcurrindent\lst@fmtindent}
2409 \gdef\lst@fmtNoindent{\global\advance\lst@fmtcurrindent-\lst@fmtindent}

```

\lst@fmtEnsureNewLine

```

2410 \gdef\lst@fmtEnsureNewLine{%
2411 \global\advance\lst@newlines\@ne
2412 \global\advance\lst@newlinesensured\@ne
2413 \lst@fmtignoretrue}

```

```

2414 \lst@AddToAtTop\lst@DoNewLines{%
2415     \ifnum\lst@newlines>\lst@newlinesensured
2416         \global\advance\lst@newlines-\lst@newlinesensured
2417     \fi
2418     \global\lst@newlinesensured\z@}
2419 \newcount\lst@newlinesensured % global
2420 \lst@AddToHook{Init}{\global\lst@newlinesensured\z@}

2421 \gdef\lst@fmtignoretrue{\let\lst@fmtifignore\iftrue}
2422 \gdef\lst@fmtignorefalse{\let\lst@fmtifignore\iffalse}
2423 \lst@AddToHook{InitVars}{\lst@fmtignorefalse}
2424 \lst@AddToHook{Output}{\lst@fmtignorefalse}

```

\lst@fmtUseLostSpace

```

2425 \gdef\lst@fmtUseLostSpace{%
2426     \lst@ifnewline \kern\lst@fmtcurrindent \global\lst@lostspace\z@
2427     \else
2428         \lst@OldOLS
2429     \fi}
2430 \lst@AddToHook{Init}
2431     {\lst@true
2432     \ifx\lst@fmtformat\@empty \ifx\lst@fmt\@empty \lst@false \fi\fi
2433     \lst@if
2434         \let\lst@OldOLS\lst@OutputLostSpace
2435         \let\lst@OutputLostSpace\lst@fmtUseLostSpace
2436         \let\lst@ProcessSpace\lst@fmtProcessSpace
2437     \fi}

```

To do: This ‘lost space’ doesn’t use \lst@alloverstyle yet!

\lst@fmtProcessSpace

```

2438 \gdef\lst@fmtProcessSpace{%
2439     \lst@ifletter
2440         \lst@Output
2441         \lst@fmtifignore\else
2442             \lst@AppendOther\lst@outputspace
2443         \fi
2444     \else \lst@ifkeepspace
2445         \lst@AppendOther\lst@outputspace
2446     \else \ifnum\lst@newlines=\z@
2447         \lst@AppendSpecialSpace
2448     \else \ifnum\lst@length=\z@
2449         \global\advance\lst@lostspace\lst@width
2450         \global\advance\lst@pos\m@ne
2451     \else
2452         \lst@AppendSpecialSpace
2453     \fi
2454     \fi \fi \fi
2455     \lst@whitespace true}

```

Formatting identifiers

\lst@fmtIdentifier We install a (keyword) test for the ‘format identifiers’.

```

2456 \lst@InstallTest{f}
2457     \lst@fmt@list\lst@fmt \lst@gfmt@list\lst@gfmt

```

```

2458 \lst@gfmt@wp
2459 wd
2460 \gdef\lst@fmt@list{\lst@fmt\lst@gfmt}\global\let\lst@fmt\@empty
2461 \gdef\lst@gfmt@list{\lst@fmt\lst@gfmt}\global\let\lst@gfmt\@empty

The working procedure expands \lst@fmt$\langle string \rangle$ (and defines \lst@PrintToken
to do nothing).

2462 \gdef\lst@gfmt@wp{%
2463 \begingroup \let\lst@UM\@empty
2464 \let\lst@PrintToken\@empty
2465 \csname\lst@ @fmt$\the\lst@token\endcsname
2466 \endgroup}

This control sequence is probably defined as ‘working identifier’.

2467 \gdef\lst@fmtIdentifier#1#2#3#4{%
2468 \lst@DefOther\lst@fmta{#2}\edef\lst@fmt{\lst@fmt,\lst@fmta}%
2469 \@namedef{\lst@ @fmt$\lst@fmta}{#3#4}}

\lst@fmt$\langle identifier \rangle$ expands to a \lst@fmtPre/\lst@fmtPost sequence defined
by #2 and #3.

2470 \lst@EndAspect
2471 </misc>

```

17.3 Line numbers

Rolf Niepraschk asked for line numbers.

```

2472 <*misc>
2473 \lst@BeginAspect{labels}

```

numbers Depending on the argument we define \lst@PlaceNumber to print the line number.

```

2474 \lst@Key{numbers}{none}{%
2475 \let\lst@PlaceNumber\@empty
2476 \lstKV@SwitchCases{#1}%
2477 {none&\\%
2478 left&\def\lst@PlaceNumber{\llap{\normalfont
2479 \lst@numberstyle{\the\lstnumber}\kern\lst@numbersep}}\\%
2480 right&\def\lst@PlaceNumber{\rlap{\normalfont
2481 \kern\linewidth \kern\lst@numbersep
2482 \lst@numberstyle{\the\lstnumber}}}%
2483 }{\PackageError{Listings}{Numbers #1 unknown}\@ehc}}

```

numberstyle Definition of the keys.

```

numbersep 2484 \lst@Key{numberstyle}{f}{\def\lst@numberstyle{#1}}
stepnumber 2485 \lst@Key{numbersep}{10pt}{\def\lst@numbersep{#1}}
numberblanklines 2486 \lst@Key{stepnumber}{1}{\def\lst@stepnumber{#1}\relax}}
numberfirstline 2487 \lst@AddToHook{EmptyStyle}{\let\lst@stepnumber\@ne}

2488 \lst@Key{numberblanklines}{true}[t]
2489 {\lstKV@SetIf{#1}\lst@ifnumberblanklines}
2490 \lst@Key{numberfirstline}{f}[t]{\lstKV@SetIf{#1}\lst@ifnumberfirstline}
2491 \gdef\lst@numberfirstlinefalse{\let\lst@ifnumberfirstline\iffalse}

```

firstnumber We select the first number according to the argument.

```

2492 \lst@Key{firstnumber}{auto}{%

```

```

2493 \lstKV@SwitchCases{#1}%
2494 {auto&\let\lst@firstnumber\@undefined\\%
2495 last&\let\lst@firstnumber\c@lstnumber
2496 }{\def\lst@firstnumber{#1\relax}}}}
2497 \lst@AddToHook{PreSet}{\let\lst@advancenum\z@}
2498 \lst@AddToHook{PreInit}
2499 {\ifx\lst@firstnumber\@undefined
2500 \let\lst@firstnumber\lst@firstline
2501 \fi}

```

\lst@SetFirstNumber Boris Veytsman proposed to continue line numbers according to listing names.

\lst@SaveFirstNumber 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`.

```

2502 \gdef\lst@SetFirstNumber{%
2503 \ifx\lst@firstnumber\@undefined
2504 \tempcnta 0\csname\@lst no\lst@intname\endcsname\relax
2505 \ifnum\@tempcnta=\z@ \tempcnta\lst@firstline
2506 \else \lst@no!oltrue \fi
2507 \advance\@tempcnta\lst@advancenum
2508 \edef\lst@firstnumber{\the\@tempcnta\relax}%
2509 \fi}

```

The current label is stored in `\lstno<name>`. If the name is empty, we use a space instead, which leaves `\lstno@` undefined.

```

2510 \gdef\lst@SaveFirstNumber{%
2511 \expandafter\xdef
2512 \csname\@lst no\ifx\lst@intname\@empty @ \else @\lst@intname\fi
2513 \endcsname{\the\c@lstnumber}}

```

\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.

```

2514 \newcounter{lstnumber}% \global
2515 \global\c@lstnumber\@ne % init
2516 \renewcommand*\thelstnumber{\@arabic\c@lstnumber}
2517 \lst@AddToHook{EveryPar}
2518 {\global\advance\c@lstnumber\lst@advancelstnum
2519 \global\advance\c@lstnumber\m@ne \refstepcounter{lstnumber}%
2520 \lst@SkipOrPrintLabel}%
2521 \global\let\lst@advancelstnum\@ne

```

Note that the counter advances *before* the label is printed and not afterwards. Otherwise we have wrong references—reported by Gregory Van Vooren.

```

2522 \lst@AddToHook{Init}{\def\@currentlabel{\thelstnumber}}

```

The label number is initialized and we ensure correct line numbers for continued listings.

```

2523 \lst@AddToHook{InitVars}
2524 {\global\c@lstnumber\lst@firstnumber
2525 \global\advance\c@lstnumber\lst@advancenum
2526 \global\advance\c@lstnumber-\lst@advancelstnum
2527 \ifx \lst@firstnumber\c@lstnumber
2528 \global\advance\c@lstnumber-\lst@advancelstnum

```

```

2529     \fi}
2530 \lst@AddToHook{ExitVars}
2531     {\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 \thelstlisting with the \ifx ... \fi construction.
2532 \AtBeginDocument{%
2533     \def\theHlstnumber{\ifx\lst@@caption\@empty \lst@neglisting
2534                                     \else \thelstlisting \fi
2535                                     .\theHlstnumber}}

```

`\lst@skipnumbers` There are more things to do. We calculate how many lines must skip their label.
The formula is

$$\text{\lst@skipnumbers} = \textit{first printing line} \bmod \text{\lst@stepnumber}.$$

Note that we use a nonpositive representative for `\lst@skipnumbers`.

```

2536 \newcount\lst@skipnumbers % \global
2537 \lst@AddToHook{Init}
2538     {\ifnum \z@>\lst@stepnumber
2539         \let\lst@advancelstnum\m@ne
2540         \edef\lst@stepnumber{-\lst@stepnumber}%
2541     \fi
2542     \ifnum \z@<\lst@stepnumber
2543         \global\lst@skipnumbers\lst@firstnumber
2544         \global\divide\lst@skipnumbers\lst@stepnumber
2545         \global\multiply\lst@skipnumbers-\lst@stepnumber
2546         \global\advance\lst@skipnumbers\lst@firstnumber
2547         \ifnum\lst@skipnumbers>\z@
2548             \global\advance\lst@skipnumbers -\lst@stepnumber
2549         \fi
    If \lst@stepnumber is zero, no line numbers are printed:
2550     \else
2551         \let\lst@SkipOrPrintLabel\relax
2552     \fi}

```

`\lst@SkipOrPrintLabel` But default is this. We use the fact that `\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 `\lst@stepnumber`.

```

2553 \gdef\lst@SkipOrPrintLabel{%
2554     \ifnum\lst@skipnumbers=\z@
2555         \global\advance\lst@skipnumbers-\lst@stepnumber\relax
2556         \lst@PlaceNumber
2557         \lst@numberfirstlinefalse
2558     \else
    If the first line of a listing should get a number, it gets it here.
2559         \lst@ifnumberfirstline
2560             \lst@PlaceNumber
2561             \lst@numberfirstlinefalse
2562         \fi
2563     \fi
2564     \global\advance\lst@skipnumbers\@ne}%

```

```

2565 \lst@AddToHook{OnEmptyLine}{%
2566     \lst@ifnumberblanklines\else \ifnum\lst@skipnumbers=\z@
2567         \global\advance\lst@skipnumbers-\lst@stepnumber\relax
2568     \fi\fi}

2569 \lst@EndAspect
2570 \</misc>

```

17.4 Line shape and line breaking

`\lst@parshape` We define a default version of `\lst@parshape` for the case that the `lineshape` aspect is not loaded. We use this `parshape` every line (in fact every paragraph). Furthermore we must repeat the `parshape` if we close a group level—or the shape is forgotten.

```

2571 \<*kernel>
2572 \def\lst@parshape{\parshape\@ne \z@ \linewidth}
2573 \lst@AddToHookAtTop{EveryLine}{\lst@parshape}
2574 \lst@AddToHookAtTop{EndGroup}{\lst@parshape}
2575 \</kernel>

```

Our first aspect in this section.

```

2576 \<*misc>
2577 \lst@BeginAspect{lineshape}

```

`xleftmargin` Usual stuff.

```

xrightmargin 2578 \lst@Key{xleftmargin}{\z@}{\def\lst@xleftmargin{#1}}
resetmargins 2579 \lst@Key{xrightmargin}{\z@}{\def\lst@xrightmargin{#1}}
               linewidth 2580 \lst@Key{resetmargins}{false}[t]{\lstKV@SetIf{#1}\lst@ifresetmargins}

```

The margins become zero if we make an exact box around the listing.

```

2581 \lst@AddToHook{BoxUnsafe}{\let\lst@xleftmargin\z@
2582                             \let\lst@xrightmargin\z@}
2583 \lst@AddToHook{TextStyle}{%
2584     \let\lst@xleftmargin\z@ \let\lst@xrightmargin\z@
2585     \let\lst@ifresetmargins\iftrue}

```

Added above hook after bug report from Magnus Lewis-Smith and José Romildo Malaquias respectively.

```

2586 \lst@Key{linewidth}\linewidth{\def\lst@linewidth{#1}}
2587 \lst@AddToHook{PreInit}{\linewidth\lst@linewidth\relax}

```

`\lst@parshape` The definition itself is easy.

```

2588 \gdef\lst@parshape{%
2589     \parshape\@ne \@totalleftmargin \linewidth}

```

We calculate the line width and (inner/outer) indent for a listing.

```

2590 \lst@AddToHook{Init}
2591     {\lst@ifresetmargins
2592         \advance\linewidth\@totalleftmargin
2593         \advance\linewidth\rightmargin
2594         \@totalleftmargin\z@
2595     \fi
2596     \advance\linewidth-\lst@xleftmargin
2597     \advance\linewidth-\lst@xrightmargin
2598     \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`.

```
2599 \lst@Key{lineskip}{\z@}{\def\lst@lineskip{#1\relax}}
2600 \lst@AddToHook{Init}
2601     {\parskip\z@
2602      \ifdim\z@=\lst@lineskip\else
2603        \@tempdima\baselineskip
2604        \advance\@tempdima\lst@lineskip
```

The following three lines simulate the ‘bad’ `\divide \@tempdima \strip@pt \baselineskip \relax`. Thanks to Peter Bartke for the bug report.

```
2605     \multiply\@tempdima\ccclvi
2606     \divide\@tempdima\baselineskip\relax
2607     \multiply\@tempdima\ccclvi
2608     \edef\baselinestretch{\strip@pt\@tempdima}%
2609     \selectfont
2610 \fi}
```

breaklines As usual we have no problems in announcing more keys. **breakatwhitespace** due to Javier Bezos.

```
breakautoindent 2611 \lst@Key{breaklines}{false}[t]{\lstKV@SetIf{#1}\lst@ifbreaklines}
breakatwhitespace 2612 \lst@Key{breakindent}{20pt}{\def\lst@breakindent{#1}}
prebreak 2613 \lst@Key{breakautoindent}{t}[t]{\lstKV@SetIf{#1}\lst@ifbreakautoindent}
postbreak 2614 \lst@Key{breakatwhitespace}{false}[t]%
2615     {\lstKV@SetIf{##1}\lst@ifbreakatwhitespace}
2616 \lst@Key{prebreak}{}{\def\lst@prebreak{#1}}
2617 \lst@Key{postbreak}{}{\def\lst@postbreak{#1}}
```

We assign some different macros and (if necessary) suppress “underfull `\hbox`” messages (and use different pretolerance):

```
2618 \lst@AddToHook{Init}
2619     {\lst@ifbreaklines
2620      \hbadness\@M \pretolerance\@M \raggedright
```

We use the normal parshape and the calculated `\lst@breakshape` (see below).

```
2621     \def\lst@parshape{\parshape\tw@ \@totalleftmargin\linewidth
2622                      \lst@breakshape}%
2623     \else
2624     \let\lst@discretionary\empty
2625     \fi}
2626 \lst@AddToHook{OnNewLine}
2627     {\lst@ifbreaklines \lst@breakNewLine \fi}
```

\lst@discretionary Here comes the whole magic: We set a discretionary break after each ‘output unit’.

\lst@spacekern 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.

```
2628 \gdef\lst@discretionary{%
2629     \lst@ifbreakatwhitespace
2630     \lst@ifwhitespace \lst@@discretionary \fi
2631     \else
2632     \lst@@discretionary
2633     \fi}%
2634 \gdef\lst@@discretionary{%
```

```

2635 \discretionary{\let\space\lst@spacekern\lst@prebreak}%
2636             {\llap{\lsthk@EveryLine
2637             \kern\lst@breakcurrindent \kern-\@totalleftmargin}%
2638             \let\space\lst@spacekern\lst@postbreak}{}}
2639 \lst@AddToHook{PostOutput}{\lst@discretionary}
2640 \gdef\lst@spacekern{\kern\lst@width}

```

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`.

`\lst@breakNewLine` We use `breakindent`, and additionally the current line indentation (coming from white spaces at the beginning of the line) if ‘auto indent’ is on.

```

2641 \gdef\lst@breakNewLine{%
2642     \@tempdima\lst@breakindent\relax
2643     \lst@ifbreakautoindent \advance\@tempdima\lst@lostspace \fi

```

Now we calculate the margin and line width of the wrapped part ...

```

2644     \@tempdimc-\@tempdima \advance\@tempdimc\linewidth
2645     \advance\@tempdima\@totalleftmargin

```

... and store it in `\lst@breakshape`.

```

2646     \xdef\lst@breakshape{\noexpand\lst@breakcurrindent \the\@tempdimc}%
2647     \xdef\lst@breakcurrindent{\the\@tempdima}}
2648 \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.

```

2649 \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.

```

2650 \gdef\lst@breakProcessOther#1{\lst@ProcessOther#1\lst@OutputOther}
2651 \lst@AddToHook{SelectCharTable}
2652     {\lst@ifbreaklines \lst@Def{'}}{\lst@breakProcessOther}}\fi}

```

A bug reported by Gabriel Tauro has been removed by using `\lst@ProcessOther` instead of `\lst@AppendOther`.

```

2653 \lst@EndAspect
2654 \</misc>

```

17.5 Frames

Another aspect.

```

2655 \<misc>
2656 \lst@BeginAspect[lineshape]{frames}

```

framexleftmargin These keys just save the argument.

```

framexrightmargin2657 \lst@Key{framexleftmargin}{\z@}{\def\lst@framexleftmargin{#1}}
framextopmargin2658 \lst@Key{framexrightmargin}{\z@}{\def\lst@framexrightmargin{#1}}
framexbottommargin2659 \lst@Key{framextopmargin}{\z@}{\def\lst@framextopmargin{#1}}
2660 \lst@Key{framexbottommargin}{\z@}{\def\lst@framexbottommargin{#1}}

```

backgroundcolor Ralf Imhäuser inspired the key `backgroundcolor`. All keys save the argument, and ...

```

2661 \lst@Key{backgroundcolor}{\def\lst@bkgcolor{#1}}
2662 \lst@Key{fillcolor}{\def\lst@fillcolor{#1}}
2663 \lst@Key{rulecolor}{\def\lst@rulecolor{#1}}
2664 \lst@Key{rulesepcolor}{\def\lst@rulesepcolor{#1}}
... some have default settings if they are empty.
2665 \lst@AddToHook{Init}{%
2666     \ifx\lst@fillcolor\@empty
2667         \let\lst@fillcolor\lst@bkgcolor
2668     \fi
2669     \ifx\lst@rulesepcolor\@empty
2670         \let\lst@rulesepcolor\lst@fillcolor
2671     \fi}

```

rulesep Another set of keys, which mainly save their respective argument. **frameshape** capitalizes all letters, and checks whether at least one round corner is specified.

framesep Eventually we define `\lst@frame` to be empty if and only if there is no frameshape.

```

frameshape2672 \lst@Key{rulesep}{2pt}{\def\lst@rulesep{#1}}
2673 \lst@Key{framerule}{.4pt}{\def\lst@framerulewidth{#1}}
2674 \lst@Key{framesep}{3pt}{\def\lst@frametextsep{#1}}
2675 \lst@Key{frameshape}{\def\lst@frameshape{#1}}
2676     \let\lst@xrulecolor\@empty
2677     \lstKV@FourArg{#1}%
2678     {\uppercase{\def\lst@frametshape{##1}}}%
2679     \uppercase{\def\lst@framelshape{##2}}}%
2680     \uppercase{\def\lst@framershape{##3}}}%
2681     \uppercase{\def\lst@framebshape{##4}}}%
2682     \let\lst@ifframeround\iffalse
2683     \lst@ifsubstring R\lst@frametshape{\let\lst@ifframeround\iffalse}%
2684     \lst@ifsubstring R\lst@framebshape{\let\lst@ifframeround\iffalse}%
2685     \def\lst@frame{##1##2##3##4}}

```

frameround We have to do some conversion here.

```

frame2686 \lst@Key{frameround}\relax
2687     {\uppercase{\def\lst@frameround{#1}}}%
2688     \expandafter\lst@frame\lst@frameround ffff\relax}
2689 \global\let\lst@frameround\@empty

```

In case of an verbose argument, we use the `trbl`-subset replacement.

```

2690 \lst@Key{frame}\relax{%
2691     \let\lst@xrulecolor\@empty
2692     \lstKV@SwitchCases{#1}%
2693     {none&\let\lst@frame\@empty}%
2694     {leftline&\def\lst@frame{l}}%
2695     {topline&\def\lst@frame{t}}%
2696     {bottomline&\def\lst@frame{b}}%

```

```

2697   lines&\def\lst@frame{tb}\\%
2698   single&\def\lst@frame{trbl}\\%
2699   shadowbox&\def\lst@frame{tRBl}%
2700       \def\lst@xrulecolor{\lst@rulesepcolor}%
2701       \def\lst@rulesep{\lst@frametextsep}%
2702   }{\def\lst@frame{#1}}%
2703   \expandafter\lstframe@\lst@frameround ffff\relax}

```

Adding t, r, b, and l in case of their upper case versions makes later tests easier.

```

2704 \gdef\lstframe@#1#2#3#4#5\relax{%
2705   \lst@ifsubstring T\lst@frame{\edef\lst@frame{t\lst@frame}}{}%
2706   \lst@ifsubstring R\lst@frame{\edef\lst@frame{r\lst@frame}}{}%
2707   \lst@ifsubstring B\lst@frame{\edef\lst@frame{b\lst@frame}}{}%
2708   \lst@ifsubstring L\lst@frame{\edef\lst@frame{l\lst@frame}}{}%

```

We now check top and bottom frame rules, ...

```

2709   \let\lst@frametshape\@empty \let\lst@framebshape\@empty
2710   \lst@frameCheck
2711       ltr\lst@framelshape\lst@frametshape\lst@framershape #4#1%
2712   \lst@frameCheck
2713       LTR\lst@framelshape\lst@frametshape\lst@framershape #4#1%
2714   \lst@frameCheck
2715       lbr\lst@framelshape\lst@framebshape\lst@framershape #3#2%
2716   \lst@frameCheck
2717       LBR\lst@framelshape\lst@framebshape\lst@framershape #3#2%

```

... look for round corners ...

```

2718   \let\lst@ifframeround\iffalse
2719   \lst@ifsubstring R\lst@frametshape{\let\lst@ifframeround\iftrue}{}%
2720   \lst@ifsubstring R\lst@framebshape{\let\lst@ifframeround\iftrue}{}%

```

and define left and right frame shape.

```

2721   \let\lst@framelshape\@empty \let\lst@framershape\@empty
2722   \lst@ifsubstring L\lst@frame
2723       {\def\lst@framelshape{YY}}%
2724       {\lst@ifsubstring l\lst@frame{\def\lst@framelshape{Y}}{}}%
2725   \lst@ifsubstring R\lst@frame
2726       {\def\lst@framershape{YY}}%
2727       {\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.

```

2728 \gdef\lst@frameCheck#1#2#3#4#5#6#7#8{%
2729   \lst@ifsubstring #1\lst@frame
2730       {\if #7T\def#4{R}\else \def#4{Y}\fi}%
2731       {\def#4{N}}%
2732   \lst@ifsubstring #3\lst@frame
2733       {\if #8T\def#6{R}\else \def#6{Y}\fi}%
2734       {\def#6{N}}%
2735   \lst@ifsubstring #2\lst@frame{\edef#5{#5#4Y#6}}{}}

```

For text style listings all frames are deactivated – added after a bug report by Stephen Reindl.

```

2736 \lst@AddToHook{TextStyle}
2737   {\let\lst@frame\@empty}

```

```

2738 \let\lst@frametshape\@empty
2739 \let\lst@framershape\@empty
2740 \let\lst@framebshape\@empty
2741 \let\lst@framelshape\@empty}

```

\lst@frameMakeVBox

```

2742 \gdef\lst@frameMakeBoxV#1#2#3{%
2743   \setbox#1\hbox{%
2744     \color@begingroup \lst@rulecolor
2745     \llap{\setbox\z@\hbox{\vrule\@width\z@\@height#2\@depth#3%
2746               \lst@frameL}%
2747           \rlap{\lst@frameBlock\lst@rulesepcolor{\wd\z@}%
2748               {\ht\z@}{\dp\z@}}}%
2749     \box\z@
2750     \ifx\lst@framelshape\@empty
2751       \kern\lst@frametextsep\relax
2752     \else
2753       \lst@frameBlock\lst@fillcolor\lst@frametextsep{#2}{#3}%
2754     \fi
2755     \kern\lst@framexleftmargin}%
2756   \rlap{\kern-\lst@framexleftmargin
2757         \@tempdima\linewidth
2758         \advance\@tempdima\lst@framexleftmargin
2759         \advance\@tempdima\lst@framexrightmargin
2760         \lst@frameBlock\lst@bkgcolor\@tempdima{#2}{#3}%
2761         \ifx\lst@framershape\@empty
2762           \kern\lst@frametextsep\relax
2763         \else
2764           \lst@frameBlock\lst@fillcolor\lst@frametextsep{#2}{#3}%
2765         \fi
2766         \setbox\z@\hbox{\vrule\@width\z@\@height#2\@depth#3%
2767               \lst@frameR}%
2768         \rlap{\lst@frameBlock\lst@rulesepcolor{\wd\z@}%
2769             {\ht\z@}{\dp\z@}}}%
2770     \box\z@}%
2771   \color@endgroup}}

```

\lst@frameBlock

```

2772 \gdef\lst@frameBlock#1#2#3#4{%
2773   \color@begingroup
2774   #1%
2775   \setbox\z@\hbox{\vrule\@height#3\@depth#4%
2776               \ifx#1\@empty \@width\z@ \kern#2\relax
2777               \else \@width#2\relax \fi}%
2778   \box\z@
2779   \color@endgroup}

```

\lst@frameR typesets right rules. We only need to iterate through \lst@framershape.

```

2780 \gdef\lst@frameR{%
2781   \expandafter\lst@frameR\lst@framershape\relax
2782   \kern-\lst@rulesep}
2783 \gdef\lst@frameR#1{%
2784   \ifx\relax#1\@empty\else
2785     \if #1Y\lst@framevrule \else \kern\lst@framerulewidth \fi

```

```

2786         \kern\lst@rulesep
2787         \expandafter\lst@frameR@b
2788     \fi}
2789 \gdef\lst@frameR@b#1{%
2790     \ifx\relax#1\@empty
2791     \else
2792         \if #1Y\color@begingroup
2793             \lst@xrulecolor
2794             \lst@framevrule
2795             \color@endgroup
2796         \else
2797             \kern\lst@framerulewidth
2798         \fi
2799     \kern\lst@rulesep
2800     \expandafter\lst@frameR@
2801 \fi}

```

\lst@frameL Ditto left rules.

```

2802 \gdef\lst@frameL{%
2803     \kern-\lst@rulesep
2804     \expandafter\lst@frameL@\lst@framelshape\relax}
2805 \gdef\lst@frameL@#1{%
2806     \ifx\relax#1\@empty\else
2807         \kern\lst@rulesep
2808         \if#1Y\lst@framevrule \else \kern\lst@framerulewidth \fi
2809     \expandafter\lst@frameL@
2810 \fi}

```

\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.

```

2811 \gdef\lst@frameH#1#2{%
2812     \global\let\lst@framediml\z@ \global\let\lst@framedimr\z@
2813     \setbox\z@\hbox{}\@tempcntb\z@
2814     \expandafter\lst@frameH@\expandafter#1#2\relax\relax\relax
2815         \@tempdimb\lst@frametextsep\relax
2816     \advance\@tempdimb\lst@framerulewidth\relax
2817         \@tempdimc-\@tempdimb
2818     \advance\@tempdimc\ht\z@
2819     \advance\@tempdimc\dp\z@
2820     \setbox\z@=\hbox{%
2821         \lst@frameHBkg\lst@fillcolor\@tempdimb\@firstoftwo
2822         \if#1T\rlap{\raise\dp\@tempboxa\box\@tempboxa}%
2823         \else\rlap{\lower\ht\@tempboxa\box\@tempboxa}\fi
2824         \lst@frameHBkg\lst@rulesepcolor\@tempdimc\@secondoftwo
2825         \advance\@tempdimb\ht\@tempboxa
2826         \if#1T\rlap{\raise\lst@frametextsep\box\@tempboxa}%
2827         \else\rlap{\lower\@tempdimb\box\@tempboxa}\fi
2828         \rlap{\box\z@}%
2829     }}
2830 \gdef\lst@frameH@#1#2#3#4{%
2831     \ifx\relax#4\@empty\else
2832         \lst@frameh \@tempcntb#1#2#3#4%
2833         \advance\@tempcntb\@ne

```

```

2834 \expandafter\lst@frameH@\expandafter#1%
2835 \fi}
2836 \gdef\lst@frameHBkg#1#2#3{%
2837 \setbox\@tempboxa\hbox{%
2838 \kern-\lst@framexleftmargin
2839 #3{\kern-\lst@framediml\relax}{\@tempdima\z@}%
2840 \ifdim\lst@framediml>\@tempdimb
2841 #3{\@tempdima\lst@framediml \advance\@tempdima-\@tempdimb
2842 \lst@frameBlock\lst@rulesepcolor\@tempdima\@tempdimb\z@}%
2843 {\kern-\lst@framediml
2844 \advance\@tempdima\lst@framediml\relax}%
2845 \fi
2846 #3{\@tempdima\z@
2847 \ifx\lst@framelshape\@empty\else
2848 \advance\@tempdima\@tempdimb
2849 \fi
2850 \ifx\lst@framershape\@empty\else
2851 \advance\@tempdima\@tempdimb
2852 \fi}%
2853 {\ifdim\lst@framedimr>\@tempdimb
2854 \advance\@tempdima\lst@framedimr\relax
2855 \fi}%
2856 \advance\@tempdima\linewidth
2857 \advance\@tempdima\lst@framexleftmargin
2858 \advance\@tempdima\lst@framexrightmargin
2859 \lst@frameBlock#1\@tempdima#2\z@
2860 #3{\ifdim\lst@framedimr>\@tempdimb
2861 \@tempdima-\@tempdimb
2862 \advance\@tempdima\lst@framedimr\relax
2863 \lst@frameBlock\lst@rulesepcolor\@tempdima\@tempdimb\z@
2864 \fi}{}%
2865 }}

```

`\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 $\{Y,N,R\} \times \{Y,N\} \times \{Y,N,R\}$. We move to the correct horizontal position, set the left corner, the horizontal line, and the right corner.

```

2866 \gdef\lst@frameh#1#2#3#4#5{%
2867 \lst@frameCalcDimA#1%
2868 \lst@ifframeround \getcirc\@tempdima \fi
2869 \setbox\z@\hbox{%
2870 \beginngroup
2871 \setbox\z@\hbox{%
2872 \kern-\lst@framexleftmargin
2873 \color@beginngroup
2874 \ifnum#1=\z@ \lst@rulecolor \else \lst@xrulecolor \fi

```

`\lst@frameCorner` gets four arguments: `\llap`, TL or BL, the corner type $\in \{Y,N,R\}$, and the size #1.

```

2875 \lst@frameCornerX\llap{#2L}#3#1%
2876 \ifdim\lst@framediml<\@tempdimb
2877 \xdef\lst@framediml{\the\@tempdimb}%

```

```

2878 \fi
2879 \begingroup
2880 \if#4Y\else \let\lst@framerulewidth\z@ \fi
2881 \tempdima\lst@framexleftmargin
2882 \advance\tempdima\lst@framexrightmargin
2883 \advance\tempdima\linewidth
2884 \vrule\@width\tempdima\@height\lst@framerulewidth \@depth\z@
2885 \endgroup
2886 \lst@frameCornerX\rlap{#2R}#5#1%
2887 \ifdim\lst@framedimr<\tempdimb
2888 \xdef\lst@framedimr{\the\tempdimb}%
2889 \fi
2890 \color@endgroup}%

2891 \if#2T\rlap{\raise\dp\z@\box\z@}%
2892 \else\rlap{\lower\ht\z@\box\z@}\fi
2893 \endgroup
2894 \box\z@}}

```

`\lst@frameCornerX` typesets a single corner and returns `\tempdimb`, the width of the corner.

```

2895 \gdef\lst@frameCornerX#1#2#3#4{%
2896 \setbox\tempboxa\hbox{\csname\lst @frame\if#3RR\fi #2\endcsname}%
2897 \tempdimb\wd\tempboxa
2898 \if #3R%
2899 #1{\box\tempboxa}%
2900 \else
2901 \if #3Y\expandafter#1\else
2902 \tempdimb\z@ \expandafter\vphantom \fi
2903 {\box\tempboxa}%
2904 \fi}

```

`\lst@frameCalcDimA` calculates an all over width; used by `\lst@frameh` and `\lst@frameInit`.

```

2905 \gdef\lst@frameCalcDimA#1{%
2906 \tempdima\lst@rulesep
2907 \advance\tempdima\lst@framerulewidth
2908 \multiply\tempdima#1\relax
2909 \advance\tempdima\lst@frametextsep
2910 \advance\tempdima\lst@framerulewidth
2911 \multiply\tempdima\tw@}

```

`\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.

```

2912 \lst@AddToHook{Init}{\lst@frameInit}
2913 \newbox\lst@framebox
2914 \gdef\lst@frameInit{%
2915 \ifx\lst@frameshape\empty \let\lst@frameL\empty \fi
2916 \ifx\lst@framershape\empty \let\lst@frameR\empty \fi
2917 \def\lst@framevrule{\vrule\@width\lst@framerulewidth\relax}%

```

We adjust values to round corners if necessary.

```

2918 \lst@ifframeround
2919 \lst@frameCalcDimA\z@ \getcirc\tempdima
2920 \tempdimb\tempdima \divide\tempdimb\tw@
2921 \advance\tempdimb -\@wholewidth
2922 \edef\lst@frametextsep{\the\tempdimb}%

```



```

2923 \edef\lst@framerulewidth{\the\@wholewidth}%
2924 \lst@frameCalcDimA\@ne \@getcirc\@tempdima
2925 \@tempdimb\@tempdima \divide\@tempdimb\tw@
2926 \advance\@tempdimb -\tw@\@wholewidth
2927 \advance\@tempdimb -\lst@frametextsep
2928 \edef\lst@rulesep{\the\@tempdimb}%
2929 \fi
2930 \lst@frameMakeBoxV\lst@framebox{\ht\strutbox}{\dp\strutbox}%
2931 \def\lst@frameLr{\copy\lst@framebox}%

```

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`?

```

2932 \ifx\lst@frametshape\@empty\else
2933 \lst@frameH T\lst@frametshape
2934 \ifvoid\z@\else
2935 \par\lst@parshape
2936 \@tempdima-\baselineskip \advance\@tempdima\ht\z@
2937 \ifdim\prevdepth<\@ccclvi\p@\else
2938 \advance\@tempdima\prevdepth
2939 \fi
2940 \ifdim\@tempdima<\z@
2941 \vskip\@tempdima\vskip\lineskip
2942 \fi
2943 \noindent\box\z@\par
2944 \lineskiplimit\maxdimen \lineskip\z@
2945 \fi
2946 \lst@frameSpreadV\lst@frametopmargin
2947 \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.

```

2948 \lst@AddToHook{EveryLine}{\lst@frameLr}
2949 \global\let\lst@frameLr\@empty

```

`\lst@frameExit` The rules at the bottom.

```

2950 \lst@AddToHook{DeInit}
2951 {\ifx\lst@framebshape\@empty\else \lst@frameExit \fi}
2952 \gdef\lst@frameExit{%
2953 \lst@frameSpreadV\lst@framexbottommargin
2954 \lst@frameH B\lst@framebshape
2955 \ifvoid\z@\else
2956 \everypar{}\par\lst@parshape\nointerlineskip\noindent\box\z@
2957 \fi}

```

`\lst@frameSpreadV` sets rules for vertical spread.

```

2958 \gdef\lst@frameSpreadV#1{%
2959 \ifdim\z@=#1\else
2960 \everypar{}\par\lst@parshape\nointerlineskip\noindent
2961 \lst@frameMakeBoxV\z@{#1}{\z@}%
2962 \box\z@
2963 \fi}

```

```

\lst@frameTR These macros make a vertical and horizontal rule. The implicit argument
\lst@frameBR \@tempdima gives the size of two corners and is provided by \lst@frameh.
\lst@frameBL 2964 \gdef\lst@frameTR{%
\lst@frameTL 2965 \vrule\@width.5\@tempdima\@height\lst@framerulewidth\@depth\z@
2966 \kern-\lst@framerulewidth
2967 \raise\lst@framerulewidth\hbox{%
2968 \vrule\@width\lst@framerulewidth\@height\z@\@depth.5\@tempdima}}
2969 \gdef\lst@frameBR{%
2970 \vrule\@width.5\@tempdima\@height\lst@framerulewidth\@depth\z@
2971 \kern-\lst@framerulewidth
2972 \vrule\@width\lst@framerulewidth\@height.5\@tempdima\@depth\z@}
2973 \gdef\lst@frameBL{%
2974 \vrule\@width\lst@framerulewidth\@height.5\@tempdima\@depth\z@
2975 \kern-\lst@framerulewidth
2976 \vrule\@width.5\@tempdima\@height\lst@framerulewidth\@depth\z@}
2977 \gdef\lst@frameTL{%
2978 \raise\lst@framerulewidth\hbox{%
2979 \vrule\@width\lst@framerulewidth\@height\z@\@depth.5\@tempdima}%
2980 \kern-\lst@framerulewidth
2981 \vrule\@width.5\@tempdima\@height\lst@framerulewidth\@depth\z@}

\lst@frameRoundT are helper macros to typeset round corners. We set height and depth to the visible
\lst@frameRoundB parts of the circle font.
2982 \gdef\lst@frameRoundT{%
2983 \setbox\@tempboxa\hbox{\@circlefnt\char\@tempcnta}%
2984 \ht\@tempboxa\lst@framerulewidth
2985 \box\@tempboxa}
2986 \gdef\lst@frameRoundB{%
2987 \setbox\@tempboxa\hbox{\@circlefnt\char\@tempcnta}%
2988 \dp\@tempboxa\z@
2989 \box\@tempboxa}

\lst@frameRTR The round corners.
\lst@frameRBR 2990 \gdef\lst@frameRTR{%
\lst@frameRBL 2991 \hb@xt@.5\@tempdima{\kern-\lst@framerulewidth
\lst@frameRTL 2992 \kern.5\@tempdima \lst@frameRoundT \hss}}
2993 \gdef\lst@frameRBR{%
2994 \hb@xt@.5\@tempdima{\kern-\lst@framerulewidth
2995 \advance\@tempcnta\@ne \kern.5\@tempdima \lst@frameRoundB \hss}}
2996 \gdef\lst@frameRBL{%
2997 \advance\@tempcnta\tw@ \lst@frameRoundB
2998 \kern-.5\@tempdima}
2999 \gdef\lst@frameRTL{%
3000 \advance\@tempcnta\thr@@\lst@frameRoundT
3001 \kern-.5\@tempdima}

3002 \lst@EndAspect
3003 </misc>

```

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.

```

\lst@makemode
\lst@ifmakekey

```

```

3004 ⟨*misc⟩
3005 \lst@BeginAspect[keywords]{make}

3006 \lst@NewMode\lst@makemode
3007 \lst@AddToHook{Output}{%
3008     \ifnum\lst@mode=\lst@makemode
3009         \ifx\lst@thestyle\lst@gkeywords@sty
3010             \lst@makekeytrue
3011         \fi
3012     \fi}

3013 \gdef\lst@makekeytrue{\let\lst@ifmakekey\iftrue}
3014 \gdef\lst@makekeyfalse{\let\lst@ifmakekey\iffalse}
3015 \global\lst@makekeyfalse % init

```

`makemacrouse` adjusts the character table if necessary

```

3016 \lst@Key{makemacrouse}f[t]{\lstKV@SetIf{#1}\lst@ifmakemacrouse}

```

`\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.

```

3017 \gdef\lst@MakeSCT{%
3018     \lst@ifmakemacrouse
3019         \lst@ReplaceInput{${}{}%
3020             \lst@PrintToken
3021             \lst@EnterMode\lst@makemode{\lst@makekeyfalse}%
3022             \lst@Merge{\lst@ProcessOther\$\lst@ProcessOther{}}%

```

The closing parenthesis tests for the mode and either processes `)` as usual or outputs it right here (in keyword style if a keyword was between `$(` and `)`).

```

3023     \lst@ReplaceInput{)}{%
3024         \ifnum\lst@mode=\lst@makemode
3025             \lst@PrintToken
3026             \begingroup
3027                 \lst@ProcessOther)%
3028                 \lst@ifmakekey
3029                     \let\lst@currstyle\lst@gkeywords@sty
3030                 \fi
3031                 \lst@OutputOther
3032             \endgroup
3033             \lst@LeaveMode
3034         \else
3035             \expandafter\lst@ProcessOther\expandafter)%
3036         \fi}%

```

If `makemacrouse` is off then both `$(` are just ‘others’.

```

3037     \else
3038         \lst@ReplaceInput{${}{\lst@ProcessOther\$\lst@ProcessOther{}}%
3039     \fi}

```

```

3040 \lst@EndAspect
3041 ⟨/misc⟩

```

18 Typesetting a listing

```

3042 (*kernel)

\lst@lineno The 'current line' counter and three keys.
  print3043 \newcount\lst@lineno % \global
firstline3044 \lst@AddToHook{InitVars}{\global\lst@lineno\@ne}
  lastline
linerange3045 \lst@Key{print}{true}[t]{\lstKV@SetIf{#1}\lst@ifprint}
3046 \lst@Key{firstline}\relax{\def\lst@firstline{#1}\relax}}
3047 \lst@Key{lastline}\relax{\def\lst@lastline{#1}\relax}}

3048 \lst@AddToHook{PreSet}
3049   {\let\lst@firstline\@ne \def\lst@lastline{9999999}\relax}}

The following code is just copied from the current development version.
3050 \lst@Key{linerange}\relax{\lstKV@OptArg[]{#1}{%
3051   \def\lst@interrange{##1}\def\lst@linerange{##2,}}}
3052 \lst@AddToHook{PreSet}{\def\lst@firstline{1}\relax}%
3053   \let\lst@linerange\@empty}
3054 \lst@AddToHook{Init}
3055 {\ifx\lst@linerange\@empty
3056   \edef\lst@linerange{{\lst@firstline}-{\lst@lastline}},}%
3057 \fi
3058 \lst@GetLineInterval}%
3059 \def\lst@GetLineInterval{\expandafter\lst@GLI\lst@linerange\@nil}
3060 \def\lst@GLI#1,#2\@nil{\def\lst@linerange{#2}\lst@GLI#1--\@nil}
3061 \def\lst@GLI#1-#2-#3\@nil{%
3062   \ifx\@empty#1\@empty
3063     \let\lst@firstline\@ne
3064   \else
3065     \def\lst@firstline{#1}\relax}%
3066 \fi
3067 \ifx\@empty#2\@empty
3068   \def\lst@lastline{9999999}\relax}%
3069 \else
3070   \ifx -#2%
3071     \let\lst@lastline\lst@firstline
3072   \else
3073     \def\lst@lastline{#2}\relax}%
3074 \fi
3075 \fi}

```

nolol is just a key here. We'll use it below, of course.

```

3076 \lst@Key{nolol}{false}[t]{\lstKV@SetIf{#1}\lst@ifnolol}
3077 \def\lst@nololtrue{\let\lst@ifnolol\iftrue}
3078 \let\lst@ifnolol\iffalse % init

```

18.1 Floats, boxes and captions

captionpos Some keys and ...

```

abovecaptionskip3079 \lst@Key{captionpos}{t}{\def\lst@captionpos{#1}}
belowcaptionskip3080 \lst@Key{abovecaptionskip}\smallskipamount{\def\lst@abovecaption{#1}}
  label3081 \lst@Key{belowcaptionskip}\smallskipamount{\def\lst@belowcaption{#1}}
  title Rolf Niepraschk proposed title.
caption3082 \lst@Key{label}\relax{\def\lst@label{#1}}

```

```

3083 \lst@Key{title}\relax{\def\lst@title{#1}\let\lst@caption\relax}
3084 \lst@Key{caption}\relax{\lstKV@OptArg[#1]{#1}%
3085     {\def\lst@caption{##2}\def\lst@caption{##1}}%
3086     \let\lst@title\@empty}
3087 \lst@AddToHookExe{TextStyle}
3088     {\let\lst@caption\@empty \let\lst@caption\@empty
3089     \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.

```

3090 \@ifundefined{thechapter}
3091     {\newcounter{lstlisting}
3092     \renewcommand\thelstlisting{\@arabic{c@lstlisting}}}
3093     {\newcounter{lstlisting}[chapter]
3094     \renewcommand\thelstlisting
3095         {\ifnum \c@chapter>\z@ \thechapter.\fi \@arabic{c@lstlisting}}}
3096 \lst@UserCommand\lstlistingname{Listing}

```

`\lst@MakeCaption` 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 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`.

```

3097 \@ifundefined{abovecaptionskip}
3098 {\newskip\abovecaptionskip
3099 \newskip\belowcaptionskip}{%
3100 \@ifundefined{\@makecaption}
3101 {\long\def\@makecaption#1#2{%
3102     \vskip\abovecaptionskip
3103     \sbox\@tempboxa{#1: #2}%
3104     \ifdim \wd\@tempboxa >\hsize
3105         #1: #2\par
3106     \else
3107         \global \@minipagefalse
3108         \hb@xt@\hsize{\hfil\box\@tempboxa\hfil}%
3109     \fi
3110     \vskip\belowcaptionskip}%
3111 }{}

```

The introduction of `\fnum@lstlisting` is also due to Axel. Previously the replacement text was used directly in `\lst@MakeCaption`. A `\noindent` has been moved elsewhere and became `\@parboxrestore` after a bug report from Frank Mittelbach.

```

3112 \def\fnum@lstlisting{%
3113     \lstlistingname
3114     \ifx\lst@caption\@empty\else~\thelstlisting\fi}%

```

We `\refstepcounter` the listing counter if and only if `\lst@caption` is not empty. Otherwise we ensure correct hyper-references, see `\lst@HRefStepCounter` below. We do this once a listing, namely at the top.

```

3115 \def\lst@MakeCaption#1{%
3116     \ifx #1t%
3117         \ifx\lst@caption\@empty\expandafter\lst@HRefStepCounter \else
3118             \expandafter\refstepcounter
3119         \fi {lstlisting}%
3120     \ifx\lst@label\@empty\else \label{\lst@label}\fi

```

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.

```

3121     \let\lst@arg\lst@intname \lst@ReplaceIn\lst@arg\lst@filenamerpl
3122     \global\let\lst@name\lst@arg \global\let\lstname\lst@name
3123     \lst@ifnolol\else
3124         \ifx\lst@caption\@empty
3125             \ifx\lst@caption\@empty
3126                 \ifx\lst@intname\@empty \else \def\lst@temp{ }%
3127                 \ifx\lst@intname\lst@temp \else
3128                     \addcontentsline{lol}{lstlisting}\lst@name
3129                 \fi\fi
3130             \fi
3131         \else
3132             \addcontentsline{lol}{lstlisting}%
3133             {\protect\numberline{\thelstlisting}\lst@caption}%
3134         \fi
3135     \fi
3136 \fi

```

We make a caption if and only if the caption is not empty and the user requested a caption at #1 ∈ {t,b}. To disallow pagebreaks between caption (or title) and a listing, we redefine the primitive \vskip locally to insert \nobreaks. Note that we allow pagebreaks in front of a ‘top-caption’ and after a ‘bottom-caption’.

To do: This redefinition is a brute force method. Is there a better one?

```

3137     \ifx\lst@caption\@empty\else
3138         \lst@ifSubstring #1\lst@captionpos
3139             {\begingroup \let\@vskip\vskip
3140                 \def\vskip{\afterassignment\lst@vskip \@tempskipa}%
3141                 \def\lst@vskip{\nobreak\@vskip\@tempskipa\nobreak}%
3142                 \par\@parboxrestore\normalsize\normalfont % \noindent (AS)
3143                 \ifx #1t\allowbreak \fi
3144                 \ifx\lst@title\@empty
3145                     \lst@makecaption\fnum@lstlisting\lst@caption % (AS)
3146                 \else
3147                     \lst@maketitle\lst@title % (AS)
3148                 \fi
3149                 \ifx #1b\allowbreak \fi
3150             \endgroup}{}%
3151     \fi}

```

I’ve inserted \normalsize after a bug report from Andreas Matthias and moved it in front of \@makecaption after receiving another from Sonja Weidmann.

`\lst@makecaption` Axel proposed the first definition. The other two are default definitions. They
`\lst@maketitle` may be adjusted to make listings compatible with other packages and classes.

```
3152 \def\lst@makecaption{\@makecaption}
3153 \def\lst@maketitle{\@makecaption\lst@title@dropdelim}
3154 \def\lst@title@dropdelim#1{\ignorespaces}
```

The following `caption(2)` support comes also from Axel.

```
3155 \AtBeginDocument{%
3156 \ifundefined{captionlabelfalse}{\def\
3157   \def\lst@maketitle{captionlabelfalse\@makecaption\@empty}}%
3158 \ifundefined{captionstartrue}{\def\
3159   \def\lst@maketitle{captionstartrue\@makecaption\@empty}}%
3160 }
```

`\lst@HRefStepCounter` 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`.

```
3161 \def\lst@HRefStepCounter#1{%
3162   \begingroup
3163   \c@lstlisting\lst@neglisting
3164   \advance\c@lstlisting\m@ne \xdef\lst@neglisting{\the\c@lstlisting}%
3165   \ifx\hyper@refstepcounter\undefined\else
3166     \hyper@refstepcounter{#1}%
3167   \fi
3168   \endgroup}
3169 \gdef\lst@neglisting{\z@}% init
```

`boxpos` sets the vertical alignment of the (possibly) used box respectively indicates that a
`\lst@boxtrue` box is used.

```
3170 \lst@Key{boxpos}{c}{\def\lst@boxpos{#1}}
3171 \def\lst@boxtrue{\let\lst@ifbox\iftrue}
3172 \let\lst@ifbox\iffalse
```

`float` Matthias Zenger asked for double-column floats, so I've inserted some code. We
`floatplacement` first check for a star ...

```
3173 \lst@Key{float}\relax[\lst@floatplacement]{%
3174   \lstKV@SwitchCases{#1}%
3175   {true&\let\lst@floatdefault\lst@floatplacement
3176     \let\lst@float\lst@floatdefault\}%
3177   false&\let\lst@floatdefault\relax
3178     \let\lst@float\lst@floatdefault
3179   }{\def\lst@next{\@ifstar{\let\lst@beginfloat\@dblfloat
3180     \let\lst@endfloat\end@dblfloat
3181     \lst@KFloat}%
3182     {\let\lst@beginfloat\@float
3183     \let\lst@endfloat\end@float
3184     \lst@KFloat}}
3185   \edef\lst@float{#1}%
3186   \expandafter\lst@next\lst@float\relax}}
```

```

... and define \lst@float.
3187 \def\lst@KFloat#1\relax{%
3188     \ifx\@empty#1\@empty
3189         \let\lst@float\lst@floatplacement
3190     \else
3191         \def\lst@float{#1}%
3192     \fi}

```

The setting `\lst@AddToHook{PreSet}{\let\lst@float\relax}` has been changed on request of Tanguy Fautré. This also led to some adjustments above.

```

3193 \lst@Key{floatplacement}{\def\lst@floatplacement{#1}}
3194 \lst@AddToHook{PreSet}{\let\lst@float\lst@floatdefault}
3195 \lst@AddToHook{TextStyle}{\let\lst@float\relax}
3196 \let\lst@floatdefault\relax % init

```

The float type `\ftype@lstlisting` is set according to whether the float package is loaded and whether `figure` and `table` floats are defined. This is done at `\begin{document}` to make the code independent of the order of package loading.

```

3197 \AtBeginDocument{%
3198     \ifundefined{c@float@type}%
3199     {\edef\ftype@lstlisting{\ifx\c@figure\@undefined 1\else 4\fi}}
3200     {\edef\ftype@lstlisting{the\c@float@type}%
3201     \addtocounter{float@type}{\value{float@type}}}%
3202 }

```

18.2 Init and EOL

aboveskip We define and initialize these keys and prevent extra spacing for ‘inline’ listings
belowskip (in particular if `fancyvrb` interface is active, problem reported by Denis Girou).

```

3203 \lst@Key{aboveskip}\medskipamount{\def\lst@aboveskip{#1}}
3204 \lst@Key{belowskip}\medskipamount{\def\lst@belowskip{#1}}
3205 \lst@AddToHook{TextStyle}
3206     {\let\lst@aboveskip\z@ \let\lst@belowskip\z@}

```

everydisplay Some things depend on display-style listings.

```

\lst@ifdisplaystyle 3207 \lst@Key{everydisplay}{\def\lst@EveryDisplay{#1}}
3208 \lst@AddToHook{TextStyle}{\let\lst@ifdisplaystyle\iffalse}
3209 \lst@AddToHook{DisplayStyle}{\let\lst@ifdisplaystyle\iftrue}
3210 \let\lst@ifdisplaystyle\iffalse

```

\lst@Init Begin a float if requested.

```

3211 \def\lst@Init#1{%
3212     \begingroup
3213     \ifx\lst@float\relax\else
3214         \edef\@tempa{\noexpand\lst@beginfloat{lstlisting}[\lst@float]}%
3215         \expandafter\@tempa
3216     \fi

```

In restricted horizontal \TeX mode we switch to `\lst@boxtrue`. In that case we make appropriate box(es) around the listing.

```

3217     \ifhmode\ifinner \lst@boxtrue \fi\fi
3218     \lst@ifbox
3219         \lsthk@BoxUnsafe
3220         \hbox to\z@\bgroup

```



```

3221      $\if t\lst@boxpos \vtop
3222      \else \if b\lst@boxpos \vbox
3223      \else \vcenter \fi\fi
3224      \bgroup \par\noindent
3225  \else
3226      \lst@ifdisplaystyle
3227      \lst@EveryDisplay
3228      \par\penalty-50\relax
3229      \vspace\lst@aboveskip
3230      \fi
3231  \fi

```

Moved `\vspace` after `\par`—or we can get an empty line atop listings. Bug reported by Jim Hefferon.

Now make the top caption.

```

3232  \normalbaselines
3233  \abovecaptionskip\lst@abovecaption\relax
3234  \belowcaptionskip\lst@belowcaption\relax
3235  \lst@MakeCaption t%

```

Some initialization. I removed `\par\nointerlineskip \normalbaselines` 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 `\parskip ≠ 0`.

```

3236  \lsthk@PreInit \lsthk@Init
3237  \lst@ifdisplaystyle
3238      \global\let\lst@ltxlabel\@empty
3239      \if@inlabel
3240          \lst@ifresetmargins
3241          \leavevmode
3242      \else
3243          \xdef\lst@ltxlabel{\the\everypar}%
3244          \lst@AddTo\lst@ltxlabel{%
3245              \global\let\lst@ltxlabel\@empty
3246              \everypar{\lsthk@EveryLine\lsthk@EveryPar}}%
3247      \fi
3248  \fi
3249  \everypar\expandafter{\lst@ltxlabel
3250                      \lsthk@EveryLine\lsthk@EveryPar}%
3251  \else
3252      \everypar{}\let\lst@NewLine\@empty
3253  \fi
3254  \lsthk@InitVars \lsthk@InitVarsBOL

```

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 `\lstenvironment@backslash`.

```

3255  \lst@Let{13}\lst@MProcessListing
3256  \let\lst@Backslash#1%
3257  \lst@EnterMode{\lst@Pmode}{\lst@SelectCharTable}%
3258  \lst@InitFinalize}

```

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`.

```

3259 \let\lst@InitFinalize\@empty % init
3260 \lst@AddToHook{PreInit}
3261     {\rightskip\z@ \leftskip\z@ \parfillskip=\z@ plus 1fil
3262       \let\par\@par}
3263 \lst@AddToHook{EveryLine}{}% init
3264 \lst@AddToHook{EveryPar}{}% init

```

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.

```

3265 \lst@Key{showlines}f[t]{\lstKV@SetIf{#1}\lst@ifshowlines}

```

\lst@DeInit Output the remaining characters and update all things. First I missed to 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.

```

3266 \def\lst@DeInit{%
3267     \lst@XPrintToken \lst@EOLUpdate
3268     \global\advance\lst@newlines\m@ne
3269     \lst@ifshowlines
3270         \lst@DoNewLines
3271     \else
3272         \setbox\@tempboxa\vbox{\lst@DoNewLines}%
3273     \fi
3274     \lst@ifdisplaystyle \par\removelastskip \fi
3275     \lsthk@ExitVars\everypar{}\lsthk@DeInit\normalbaselines

```

Place the bottom caption.

```

3276     \lst@MakeCaption b%

```

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.

```

3277     \lst@ifbox
3278         \egroup $\hss \egroup
3279         \vrule\@width\lst@maxwidth\@height\z@\@depth\z@
3280     \else
3281         \lst@ifdisplaystyle
3282             \par\penalty-50\vspace\lst@belowskip
3283         \fi
3284     \fi

```

End the float if necessary.

```

3285     \ifx\lst@float\relax\else
3286         \expandafter\lst@endfloat
3287     \fi
3288     \endgroup}

```

\lst@maxwidth is to be allocated, initialized and updated.

```

3289 \newdimen\lst@maxwidth % \global
3290 \lst@AddToHook{InitVars}{\global\lst@maxwidth\z@}
3291 \lst@AddToHook{InitVarsEOL}
3292     {\ifdim\lst@currlwidth>\lst@maxwidth
3293       \global\lst@maxwidth\lst@currlwidth
3294     \fi}

```

`\lst@EOLUpdate` What do you think this macro does?

```
3295 \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.

```
3296 \def\lst@MProcessListing{%
3297   \lst@XPrintToken \lst@EOLUpdate \lsthk@InitVarsBOL
3298   \global\advance\lst@lineno\@ne
3299   \ifnum \lst@lineno>\lst@lastline
3300     \lst@ifdropinput \lst@LeaveMode \fi
3301     \ifx\lst@linerange\@empty
3302       \expandafter\expandafter\expandafter\lst@EndProcessListing
3303     \else
3304       \lst@interrange
3305       \lst@GetLineInterval
3306       \expandafter\expandafter\expandafter\lst@SkipToFirst
3307     \fi
3308   \else
3309     \expandafter\lst@BOLGobble
3310   \fi}
```

`\lst@EndProcessListing` Default definition is `\endinput`. This works for `\lstinputlisting`.

```
3311 \let\lst@EndProcessListing\endinput
```

`gobble` The key sets the number of characters to gobble each line.

```
3312 \lst@Key{gobble}{0}{\def\lst@gobble{#1}}
```

`\lst@BOLGobble` If the number is positive, we set a temporary counter and start a loop.

```
3313 \def\lst@BOLGobble{%
3314   \ifnum\lst@gobble>\z@
3315     \@tempcnta\lst@gobble\relax
3316     \expandafter\lst@BOLGobble@
3317   \fi}
```

A nonpositive number terminates the loop (by not continuing). Note: This is not the macro just used in `\lst@BOLGobble`.

```
3318 \def\lst@BOLGobble@@{%
3319   \ifnum\@tempcnta>\z@
3320     \expandafter\lst@BOLGobble@
3321   \fi}
```

If we gobble a backslash, we have to look whether this backslash ends an environment. Whether the coming characters equal e.g. `end{lstlisting}`, we either end the environment or insert all just eaten characters after the 'continue loop' macro.

```
3322 \def\lstenv@BOLGobble@@{%
3323   \lst@ifnextchars\lstenv@endstring{\lstenv@End}%
3324   {\advance\@tempcnta\m@ne \expandafter\lst@BOLGobble@@\lst@eaten}}
```

Now comes the loop: if we read `\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 `\lst@ProcessFormFeed`. This was a bug analyzed by Heiko Oberdiek.

```
3325 \def\lst@BOLGobble@#1{%
3326   \let\lst@next#1%
```

```

3327 \ifx \lst@next\relax\else
3328 \ifx \lst@next\lst@MProcessListing\else
3329 \ifx \lst@next\lst@processformfeed\else

    Otherwise we use one of the two submacros.

3330 \ifx \lst@next\lstenv@backslash
3331 \let\lst@next\lstenv@BOLGobble@@
3332 \else
3333 \let\lst@next\lst@BOLGobble@@

    Now we really gobble characters. A tabulator decreases the temporary counter by
    \lst@tabsize (and deals with remaining amounts, if necessary), ...

3334 \ifx #1\lst@processtabulator
3335 \advance\@tempcnta-\lst@tabsize\relax
3336 \ifnum\@tempcnta<\z@
3337 \lst@length-\@tempcnta \lst@PreGotoTabStop
3338 \fi

    ... whereas any other character decreases the counter by one.

3339 \else
3340 \advance\@tempcnta\m@ne
3341 \fi
3342 \fi \fi \fi \fi
3343 \lst@next}

3344 \def\lst@processformfeed{\lst@ProcessFormFeed}
3345 \def\lst@processtabulator{\lst@ProcessTabulator}

```

18.3 List of listings

`\name` Each pretty-printing command values `\lst@intname` before setting any keys.

```

\lstname 3346 \lst@Key{name}\relax{\def\lst@intname{#1}}
\lst@name 3347 \lst@AddToHookExe{PreSet}{\global\let\lst@intname\@empty}
\lst@intname 3348 \lst@AddToHook{PreInit}{%
3349 \let\lst@arg\lst@intname \lst@ReplaceIn\lst@arg\lst@filenamerpl
3350 \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.

```

3351 \def\lst@filenamerpl{\_ \textunderscore $\textdollar -\textendash}

```

`\l@lstlisting` prints one ‘lol’ line.

```

3352 \def\l@lstlisting#1#2{\@dottedtocline{1}{1.5em}{2.3em}{#1}{#2}}

```

`\lstlistlistingname` contains simply the header name.

```

3353 \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.

```

3354 \lst@UserCommand\lstlistoflistings{\bgroup
3355 \let\contentsname\lstlistlistingname
3356 \let\lst@temp\starttoc \def\starttoc#1{\lst@temp{lol}}%
3357 \tableofcontents \egroup}

```

For KOMA-script classes, we define it a la KOMA thanks to a bug report by Tino Langer.

```

3358 \@ifpackageloaded{scrfile}
3359 {\newcommand*\lol@heading{\float@listhead{\lstlistlistingname}}
3360 \renewcommand*\lstlistoflistings{%
3361   \begingroup%
3362     \if@twocolumn
3363       \@restonecoltrue\onecolumn
3364     \else
3365       \@restonecolfalse
3366     \fi
3367     \lol@heading%
3368     \@parskipfalse\@parskip@indent%
3369     \@starttoc{lol}%
3370     \if@restonecol\twocolumn\fi
3371   \endgroup}%
3372 }{}

```

18.4 Inline listings

`\lstinline` In addition to `\lsthk@PreSet`, 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.

```

3373 \newcommand\lstinline[1][]{%
3374   \leavevmode\bgroup % \hbox\bgroup --> \bgroup
3375   \def\lst@boxpos{b}%
3376   \lsthk@PreSet\lstset{flexiblecolumns,#1}%
3377   \lsthk@TextStyle
3378   \@ifnextchar\bgroup{\afterassignment\lst@InlineG \let\@let@token}%
3379   \lstinline@}
3380 \def\lstinline@#1{%
3381   \lst@Init\relax
3382   \lst@ifnextcharactive{\lst@InlineM#1}{\lst@InlineJ#1}}
3383 \lst@AddToHook{TextStyle}{}% init
3384 \lst@AddToHook{SelectCharTable}{\lst@inlinechars}
3385 \global\let\lst@inlinechars\@empty

```

`\lst@InlineM` treat the cases of 'normal' inlines and inline listings inside an argument. In the `\lst@InlineJ` first case the given character ends the inline listing and EOL within such a listing immediately ends it and produces an error message.

```

3386 \def\lst@InlineM#1{\gdef\lst@inlinechars{%
3387   \lst@Def{'#1}{\lst@DeInit\egroup\global\let\lst@inlinechars\@empty}%
3388   \lst@Def{13}{\lst@DeInit\egroup\global\let\lst@inlinechars\@empty
3389     \PackageError{Listings}{\lstinline ended by EOL}\@ehc}}%
3390 \lst@inlinechars}

```

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.

```

3391 \def\lst@InlineJ#1{%

```

```

3392 \def\lst@temp##1#1{%
3393     \let\lst@arg\@empty \lst@InsideConvert{##1}\lst@arg
3394     \lst@DeInit\egroup}%
3395 \lst@temp}

```

`\lst@InlineG` is experimental.

```

3396 \def\lst@InlineG{%
3397     \lst@Init\relax
3398     \lst@ifnextcharActive{\lst@InlineM\}}%
3399     {\let\lst@arg\@empty \lst@InlineGJ}}
3400 \def\lst@InlineGJ{\futurelet\@let@token\lst@InlineGJTest}%
3401 \def\lst@InlineGJTest{%
3402     \ifx\@let@token\egroup
3403         \afterassignment\lst@InlineGJEnd
3404         \expandafter\let\expandafter\@let@token
3405     \else
3406         \ifx\@let@token\@sptoken
3407             \let\lst@next\lst@InlineGJReadSp
3408         \else
3409             \let\lst@next\lst@InlineGJRead
3410         \fi
3411         \expandafter\lst@next
3412     \fi}
3413 \def\lst@InlineGJEnd{\lst@arg\lst@DeInit\egroup}
3414 \def\lst@InlineGJRead#1{%
3415     \lccode'\~='#1\lowercase{\lst@lAddTo\lst@arg~}%
3416     \lst@InlineGJ}
3417 \def\lst@InlineGJReadSp#1{%
3418     \lccode'\~=' \lowercase{\lst@lAddTo\lst@arg~}%
3419     \lst@InlineGJ#1}

```

18.5 The input command

`\lstinputlisting` inputs the listing or asks the user for a new file name.

```

3420 \def\lstinputlisting{%
3421     \begingroup \lst@setcatcodes \lst@inputlisting}
3422 \newcommand\lst@inputlisting[2][]{%
3423     \endgroup
3424     \def\lst@set{#1}%
3425     \IfFileExists{#2}%
3426         {\lst@InputListing{#2}}%
3427         {\filename@parse{#2}%
3428         \edef\reserved@a{\noexpand\lst@MissingFileError
3429             {\filename@area\filename@base}%
3430             {\ifx\filename@ext\relax tex\else\filename@ext\fi}}%
3431         \reserved@a}%
3432     \@doendpe \@newlistfalse \ignorespaces}

```

We use `\@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 L^AT_EX's `\@missingfileerror`. The parenthesis have been added after Heiko Oberdiek reported about a problem discussed on TEX-D-L.

```

3433 \def\lst@MissingFileError#1#2{%
3434   \typeout{^^J! Package Listings Error: File ‘#1(.#2)’ not found.^^J%
3435   ^^JType X to quit or <RETURN> to proceed,^^J%
3436   or enter new name. (Default extension: #2)^^J}%
3437   \message{Enter file name: }%
3438   {\newlinechar\m@ne \global\read\m@ne to\@gtempa}%

   Typing x or X exits.
3439   \ifx\@gtempa\@empty \else
3440     \def\reserved@a{x}\ifx\reserved@a\@gtempa\batchmode\@@end\fi
3441     \def\reserved@a{X}\ifx\reserved@a\@gtempa\batchmode\@@end\fi

   In all other cases we try the new file name.
3442     \filename@parse\@gtempa
3443     \edef\filename@ext{%
3444       \ifx\filename@ext\relax#2\else\filename@ext\fi}%
3445     \edef\reserved@a{\noexpand\IfFileExists %
3446       {\filename@area\filename@base.\filename@ext}%
3447       {\noexpand\lst@InputListing %
3448       {\filename@area\filename@base.\filename@ext}}}%
3449     {\noexpand\lst@MissingFileError
3450     {\filename@area\filename@base}{\filename@ext}}}%
3451     \expandafter\reserved@a %
3452     \fi}

```

`\lst@ifdraft` makes use of `\lst@ifprint`. Enrico Straube requested the final option.

```

3453 \let\lst@ifdraft\iffalse
3454 \DeclareOption{draft}{\let\lst@ifdraft\iftrue}
3455 \DeclareOption{final}{\let\lst@ifdraft\iffalse}
3456 \lst@AddToHook{PreSet}
3457   {\lst@ifdraft
3458     \let\lst@ifprint\iffalse
3459     \@gobbletwo\fi\fi
3460   \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.

```

3461 \def\lst@InputListing#1{%
3462   \begingroup
3463     \lsthk@PreSet \gdef\lst@intname{#1}%
3464     \expandafter\lstset\expandafter{\lst@set}%
3465     \lsthk@DisplayStyle
3466     \catcode\active=\active
3467     \lst@Init\relax \let\lst@gobble\z@
3468     \lst@SkipToFirst
3469     \lst@ifprint \def\lst@next{\input{#1}}%
3470     \else \let\lst@next\@empty \fi
3471     \lst@next
3472     \lst@DeInit
3473   \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.

```

3474 \def\lst@SkipToFirst{%
3475     \ifnum \lst@lineno<\lst@firstline
        We drop the input and redefine the end of line characters.
3476         \lst@BeginDropInput\lst@Pmode
3477         \lst@Let{13}\lst@MSkipToFirst
3478         \lst@Let{10}\lst@MSkipToFirst
3479     \else
3480         \expandafter\lst@BOLGobble
3481     \fi}

```

`\lst@MSkipToFirst` We just look whether to drop more lines or to leave the mode which restores the definition of `chr(13)` and `chr(10)`.

```

3482 \def\lst@MSkipToFirst{%
3483     \global\advance\lst@lineno\@ne
3484     \ifnum \lst@lineno=\lst@firstline
3485         \lst@LeaveMode \global\lst@newlines\z@
3486         \lsthk@InitVarsBOL
3487         \expandafter\lst@BOLGobble
3488     \fi}

```

18.6 The environment

18.6.1 Low-level processing

`\lstenv@DroppedWarning` gives a warning if characters have been dropped.

```

3489 \def\lstenv@DroppedWarning{%
3490     \ifx\lst@dropped\undefined\else
3491         \PackageWarning{Listings}{Text dropped after begin of listing}%
3492     \fi}
3493 \let\lst@dropped\undefined % init

```

`\lstenv@Process` We execute ‘`\lstenv@ProcessM`’ or `\lstenv@ProcessJ` according to whether we find an active EOL or a nonactive `^^J`.

```

3494 \begingroup \lccode\~='^^M\lowercase{%
3495 \gdef\lstenv@Process#1{%
3496     \ifx~#1%

```

We make no extra `\lstenv@ProcessM` definition since there is nothing to do at all if we’ve found an active EOL.

```

3497         \lstenv@DroppedWarning \let\lst@next\lst@SkipToFirst
3498     \else\ifx^^J#1%
3499         \lstenv@DroppedWarning \let\lst@next\lstenv@ProcessJ
3500     \else
3501         \let\lst@dropped#1\let\lst@next\lstenv@Process
3502     \fi \fi
3503     \lst@next}
3504 }\endgroup

```

`\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’.

```

3505 \def\lstenv@ProcessJ{%
3506     \let\lst@arg\@empty

```



```

3507 \ifx\@currenvir\lstenv@name
3508 \expandafter\lstenv@ProcessJEnv
3509 \else

```

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 `\lstenv@ProcessJ` we call the temporary macro after expanding `\fi`.

```

3510 \expandafter\def\expandafter\lst@temp\expandafter##1%
3511 \csname end\lstenv@name\endcsname
3512 {\lst@InsideConvert{##1}\lstenv@ProcessJ}%
3513 \expandafter\lst@temp
3514 \fi}

```

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)=^^J` now and not `chr(13)`.

```

3515 \begingroup \lccode'\~='\\lowercase{%
3516 \gdef\lstenv@ProcessJ@{%
3517 \lst@lExtend\lst@arg
3518 {\expandafter\ \expandafter~\lstenv@endstring}%
3519 \catcode10=\active \lst@Let{10}\lst@MProcessListing

```

We execute `\lst@arg` to typeset the listing.

```

3520 \lst@SkipToFirst \lst@arg}
3521 }\endgroup

```

`\lstenv@ProcessJEnv` 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.

```

3522 \def\lstenv@ProcessJEnv#1\end#2{\def\lst@temp{#2}%
3523 \ifx\lstenv@name\lst@temp
3524 \lst@InsideConvert{#1}%
3525 \expandafter\lstenv@ProcessJ@
3526 \else

```

Otherwise we append the characters including the eaten `\end` and the eaten argument to current `\lst@arg`. And we look for the end of environment again.

```

3527 \lst@InsideConvert{#1\\end\{#2\}}%
3528 \expandafter\lstenv@ProcessJEnv
3529 \fi}

```

`\lstenv@backslash` Coming to a backslash we either end the listing or process a backslash and insert the eaten characters again.

```

3530 \def\lstenv@backslash{%
3531 \lst@ifNextChars\lstenv@endstring
3532 {\lstenv@End}%
3533 {\expandafter\lst@backslash \lst@eaten}}%

```

`\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.

```

3534 \def\lstenv@End{%
3535 \ifx\@currenvir\lstenv@name
3536 \edef\lst@next{\noexpand\end\lstenv@name}}%

```

```

3537 \else
3538 \def\lst@next{\csname end\lstenv@name\endcsname}%
3539 \fi
3540 \lst@next}

```

18.6.2 \lstnewenvironment

`\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 TeXbook’.

```

3541 \lst@UserCommand\lstnewenvironment#1#2#{%
3542 \ifundefined{#1}%
3543 {\let\lst@arg\@empty
3544 \lst@XConvert{#1}\@nil
3545 \expandafter\lstnewenvironment@\lst@arg{#1}{#2}}%
3546 {\PackageError{Listings}{Environment ‘#1’ already defined}\@eha
3547 \gobbletwo}}
3548 \def\@tempa#1#2#3{%
3549 \gdef\lstnewenvironment@##1##2##3##4##5{%
3550 \begingroup

```

A lonely ‘end environment’ produces an error.

```

3551 \global\@namedef{end##2}{\lstenv@Error{##2}}%

```

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.

```

3552 \global\@namedef{##2}{\def\lstenv@name{##2}%
3553 \begingroup \lst@setcatcodes \catcode\active=\active
3554 \csname##2\endcsname}%

```

The submacro is defined via `\new@command`. We misuse `\l@ngrel@x` to make the definition `\global` and refine L^AT_EX’s `\@xargdef`.

```

3555 \let\l@ngrel@x\global
3556 \let\@xargdef\lstenv@xargdef
3557 \expandafter\new@command\csname##2\endcsname##3%

```

First we execute `##4=(begin code)`. Then follows the definition of the terminating string (`end{lstlisting}` or `endlstlisting`, for example):

```

3558 {\lsthk@PreSet ##4%
3559 \ifx\@currenvir\lstenv@name
3560 \def\lstenv@endstring{#1#2##1#3}%
3561 \else
3562 \def\lstenv@endstring{#1##1}%
3563 \fi

```

We redefine (locally) ‘end environment’ since ending is legal now. Note that the redefinition also works inside a TeX comment line.

```

3564 \@namedef{end##2}{\lst@DeInit ##5\endgroup
3565 \doendpe \@ignoretrue}%

```

`\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.

```

3566      \lsthk@DisplayStyle
3567      \let\lst@EndProcessListing\lstenv@SkipToEnd
3568      \lst@Init\lstenv@backslash
3569      \lst@ifprint
3570          \expandafter\expandafter\expandafter\lstenv@Process
3571      \else
3572          \expandafter\lstenv@SkipToEnd
3573      \fi
3574      \lst@insertargs}%
3575  \endgroup}%
3576 }
3577 \let\lst@arg\@empty \lst@XConvert{end}\{\}\@nil
3578 \expandafter\@tempa\lst@arg
3579 \let\lst@insertargs\@empty

```

`\lstenv@xargdef` This is a derivation of L^AT_EX's `\@xargdef`. We expand the submacro's name, use `\gdef` instead of `\def`, and hard code a kind of `\@protected@testopt`.

```

3580 \def\lstenv@xargdef#1{
3581     \expandafter\lstenv@xargdef@\csname\string#1\endcsname#1}
3582 \def\lstenv@xargdef@#1#2[#3][#4]#5{%
3583     \@ifdefinable#2{%
3584         \gdef#2{%
3585             \ifx\protect\@typeset@protect
3586                 \expandafter\lstenv@testopt
3587             \else
3588                 \@x@protect#2%
3589             \fi
3590             #1%
3591             {#4}}}%
3592     \@yargdef
3593     #1%
3594     \tw@
3595     {#3}%
3596     {#5}}

```

`\lstenv@testopt` The difference between this macro and `\@testopt` is that we temporarily reset the catcode of the EOL character `^M` to read the optional argument.

```

3597 \long\def\lstenv@testopt#1#2{%
3598     \@ifnextchar[{\catcode\active5\relax \lstenv@testopt@#1}%
3599     {#1[#2]}}
3600 \def\lstenv@testopt@#1[#2]{%
3601     \catcode\active\active
3602     #1[#2]}

```

`\lstenv@SkipToEnd` We use the temporary definition

```

\long\def\lst@temp##1\<content of \lstenv@endstring\{\lstenv@End}

```

which gobbles all characters up to the end of environment and finishes it.

```

3603 \begingroup \lccode'\~='\\lowercase{%
3604 \gdef\lstenv@SkipToEnd{%
3605     \long\expandafter\def\expandafter\lst@temp\expandafter##\expandafter
3606     1\expandafter~\lstenv@endstring{\lstenv@End}%
3607     \lst@temp}
3608 }\endgroup

```

`\lstenv@Error` is called by a lonely ‘end environment’.

```
3609 \def\lstenv@Error#1{\PackageError{Listings}{Extra \string\end#1}%
3610      {I’m ignoring this, since I wasn’t doing a \csname#1\endcsname.}}
```

`\lst@TestEOLChar` Here we test for the two possible EOL characters.

```
3611 \begingroup \lccode‘\~‘\^^M\lowercase{%
3612 \gdef\lst@TestEOLChar#1{%
3613     \def\lst@insertargs{#1}%
3614     \ifx ~#1\@empty \else
3615     \ifx ^^J#1\@empty \else
3616         \global\let\lst@intname\lst@insertargs
3617         \let\lst@insertargs\@empty
3618     \fi \fi}
3619 }\endgroup
```

`\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.

```
3620 \lstnewenvironment{lstlisting}[2] []
3621     {\lst@TestEOLChar{#2}%
3622     \lstset{#1}%
3623     \csname\@lst @SetFirstNumber\endcsname}
3624     {\csname\@lst @SaveFirstNumber\endcsname}
3625 \</kernel>
```

19 Documentation support

```
\begin{lstsample}{\langle point list \rangle}{\langle left \rangle}{\langle right \rangle}
\end{lstsample}
```

Roughly speaking all material in between this environment is executed ‘on the left side’ and typeset verbatim on the right. `\langle left \rangle` is executed before the left side is typeset, and similarly `\langle right \rangle` before the right-hand side.

`\langle point list \rangle` is used as argument to the `point` key. This is a special key used to highlight the keys in the examples.

```
\begin{lstxsample}{\langle point list \rangle}
\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.

```
\newdocenvironment{\langle name \rangle}{\langle short name \rangle}{\langle begin code \rangle}{\langle end code \rangle}
```

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<list macro>{\<input files>}{\<don't input>}`

scans `{\<input files>}\{\<don't input>}` for language definitions. The available languages are stored in `<list macro>` using the form `<language>(<dialtect>),.`

`\lstprintlanguages<list macro>`

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.

```
3626 <*doc>
3627 \let\lstdoc@currversion\fileversion
3628 \RequirePackage[writefile]{listings}[2002/04/01]
3629 \newif\iffancyvrb \IfFileExists{fancyvrb.sty}{\fancyvrbtrue}{\fancyvrbfalse}
3630 \newif\ifcolor \IfFileExists{color.sty}{\colortrue}{\colorfalse}
3631 \newif\ifhyper \@ifundefined{pdfoutput}{\ifhyperfalse}{\ifhypertrue}
3632 \IfFileExists{hyperref.sty}{\hypertrue}{\hyperfalse}
3633 \newif\ifalgorithmic \IfFileExists{algorithmic.sty}{\algorithmictrue}{\algorithmicfalse}
3634 \newif\iflgrind \IfFileExists{lgrind.sty}{\lgrindtrue}{\lgrindfalse}
3635 \iffancyvrb \RequirePackage{fancyvrb}\fi
3636 \ifhyper \RequirePackage[colorlinks]{hyperref}\else
3637 \def\href#1{\texttt{}}\fi
3638 \ifcolor \RequirePackage{color}\fi
3639 \ifalgorithmic \RequirePackage{algorithmic}\fi
3640 \iflgrind \RequirePackage{lgrind}\fi
3641 \RequirePackage{nameref}
3642 \RequirePackage{url}
3643 \renewcommand\ref{\protect\T@ref}
3644 \renewcommand\pageref{\protect\T@pageref}
```

19.2 Environments for notes

`\lst@BeginRemark` We begin with two simple definitions ...

`\lst@EndRemark` 3645 `\def\lst@BeginRemark#1{%`

3646 `\begin{quote}\topsep0pt\let\small\footnotesize\small#1:}`

3647 `\def\lst@EndRemark{\end{quote}}`

TODO ... used to define some environments.

ALTERNATIVE 3648 `\newenvironment{TODO}`

REMOVED 3649 `{\lst@BeginRemark{To do}}{\lst@EndRemark}`

OLDDEF 3650 `\newenvironment{ALTERNATIVE}`

```

3651     {\lst@BeginRemark{Alternative}}{\lst@EndRemark}
3652 \newenvironment{REMOVED}
3653     {\lst@BeginRemark{Removed}}{\lst@EndRemark}
3654 \newenvironment{OLDDEF}
3655     {\lst@BeginRemark{Old definition}}{\lst@EndRemark}

  advise  The environment uses \@listi.
\advisespace 3656 \def\advise{\par\list\labeladvise
3657     {\advance\linewidth\@totalleftmargin
3658       \@totalleftmargin\z@
3659       \@listi
3660       \let\small\footnotesize \small\sffamily
3661       \parsep \z@ \@plus\z@ \@minus\z@
3662       \topsep6\p@ \@plus1\p@ \@minus2\p@
3663       \def\makelabel##1{\hss\llap{##1}}}}
3664 \let\endadvise\endlist

3665 \def\advisespace{\hbox{}}\qquad}
3666 \def\labeladvise{$\to$}

  syntax  This environment uses \list with a special \makelabel, ...
\syntaxbreak 3667 \newenvironment{syntax}
\syntaxnewline 3668     {\list{}{\itemindent-\leftmargin
\syntaxor 3669       \def\makelabel##1{\hss\lst@syntaxlabel##1,,,\relax}}}
3670     {\endlist}

    ... which is defined here. The comma separated items are placed as needed.
3671 \def\lst@syntaxlabel#1,#2,#3,#4\relax{%
3672     \llap{\scriptsize\itshape#3}%
3673     \def\lst@temp{#2}%
3674     \expandafter\lst@syntaxlabel@\meaning\lst@temp\relax
3675     \rlap{\hskip-\itemindent\hskip\itemsep\hskip\linewidth
3676         \llap{\ttfamily\lst@temp}\hskip\labelwidth
3677         \def\lst@temp{#1}%
3678         \ifx\lst@temp\lstdoc@currversion#1\fi}}
3679 \def\lst@syntaxlabel@#1>#2\relax
3680     {\edef\lst@temp{\zap@space#2 \@empty}}

3681 \newcommand*\syntaxnewline{\newline\hbox{}}\kern\labelwidth}
3682 \newcommand*\syntaxor{\qquad or\qquad}
3683 \newcommand*\syntaxbreak
3684     {\hfill\kernOpt\discretionary{}{\kern\labelwidth}{}}
3685 \let\syntaxfill\hfill

\alternative iterates down the list and inserts vertical rule(s).
3686 \def\alternative#1{\lst@true \alternative@#1,\relax,}
3687 \def\alternative@#1,{%
3688     \ifx\relax#1\@empty
3689         \expandafter\@gobble
3690     \else
3691         \ifx\@empty#1\@empty\else
3692             \lst@if \lst@false \else $\vert$\fi
3693             \textup{\texttt{#1}}}%
3694         \fi
3695     \fi
3696     \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 `<short name>`.

```

3697 \long\def\m@cro@#1#2#3{\endgroup \topsep\MacroTopsep \trivlist
3698   \edef\saved@macroname{\string#3}%
3699   \def\makelabel##1{\llap{##1}}%
3700   \if@inlabel
3701     \let\@tempa\@empty \count@\macro@cnt
3702     \loop \ifnum\count@>\z@
3703       \edef\@tempa{\@tempa\hbox{\strut}}\advance\count@\m@ne \repeat
3704     \edef\makelabel##1{\llap{\vtop to\baselineskip
3705                           {\@tempa\hbox{##1}\vss}}}%
3706     \advance \macro@cnt \@ne
3707   \else \macro@cnt\@ne \fi
3708   \edef\@tempa{\noexpand\item[%
3709     #1%
3710     \noexpand\PrintMacroName
3711     \else

```

The next line has been modified.

```

3712     \expandafter\noexpand\csname Print#2Name\endcsname % MODIFIED
3713     \fi
3714     {\string#3}}}%
3715   \@tempa
3716   \global\advance\c@CodelineNo\@ne
3717   #1%
3718   \SpecialMainIndex{#3}\nobreak
3719   \DoNotIndex{#3}%
3720   \else

```

Ditto.

```

3721     \csname SpecialMain#2Index\endcsname{#3}\nobreak % MODIFIED
3722     \fi
3723     \global\advance\c@CodelineNo\m@ne
3724     \ignorespaces}

```

`\macro` These two definitions need small adjustments due to the modified `\m@cro@`.

```

\environment 3725 \def\macro{\begingroup
3726   \catcode'\12
3727   \MakePrivateLetters \m@cro@ \iftrue {Macro}}% MODIFIED
3728 \def\environment{\begingroup
3729   \catcode'\12
3730   \MakePrivateLetters \m@cro@ \iffalse {Env}}% MODIFIED

```

`\newdocenvironment` This command simply makes definitions similar to `'environment'` and provides the printing and indexing commands.

```

3731 \def\newdocenvironment#1#2#3#4{%
3732   \@namedef{#1}{#3\begingroup \catcode'\12\relax
3733     \MakePrivateLetters \m@cro@ \iffalse {#2}}%
3734   \@namedef{end#1}{#4\endmacro}%
3735   \@ifundefined{Print#2Name}{\expandafter
3736     \let\csname Print#2Name\endcsname\PrintMacroName}{}}%

```

```

3737 \ifundefined{SpecialMain#2Index}{\expandafter
3738 \let\csname SpecialMain#2Index\endcsname\SpecialMainIndex}{}}

aspect The environment and its ‘print’ and ‘index’ commands.
\PrintAspectName 3739 \newdocenvironment{aspect}{Aspect}{}{}
\SpecialMainAspectIndex 3740 \def\PrintAspectName#1{
3741 \def\SpecialMainAspectIndex#1{%
3742 \@bsphack
3743 \index{aspects:\levelchar\protect\aspectname{#1}\encapchar main}%
3744 \@esphack}

lstkey One more environment with its ‘print’ and ‘index’ commands.
\PrintKeyName 3745 \newdocenvironment{lstkey}{Key}{}{}
\SpecialMainKeyIndex 3746 \def\PrintKeyName#1{\strut\keyname{#1}\ }
3747 \def\SpecialMainKeyIndex#1{%
3748 \@bsphack
3749 \index{keys\levelchar\protect\keyname{#1}\encapchar main}%
3750 \@esphack}

\labelargcount We just allocate a counter and use LATEX’s \list to implement this environment.
macroargs 3751 \newcounter{argcount}
3752 \def\labelargcount{\texttt{\#\arabic{argcount}}\hspace\labelsep$=$}
3753 \def\macroargs{\list\labelargcount
3754 {\usecounter{argcount}\leftmargin=2\leftmargin
3755 \parsep \z@ \@plus\z@ \@minus\z@
3756 \topsep4\p@ \@plus\p@ \@minus2\p@
3757 \itemsep\z@ \@plus\z@ \@minus\z@
3758 \def\makelabel##1{\hss\llap{##1}}}}
3759 \def\endmacroargs{\endlist\@endparenv}

```

19.4 The lstsample environment

lstsample We store the verbatim part and write the source code also to file.

```

3760 \lst@RequireAspects{writefile}
3761 \newbox\lst@samplebox
3762 \lstnewenvironment{lstsample}[3] []
3763 {\global\let\lst@intname\@empty
3764 \gdef\lst@sample{#2}%
3765 \setbox\lst@samplebox=\hbox\bgroup
3766 \setkeys{lst}{language={},style={},tabsize=4,gobble=5,%
3767 basicstyle=\small\ttfamily,basewidth=0.51em,point={#1}}
3768 #3%
3769 \lst@BeginAlsoWriteFile{\jobname.tmp}}
3770 {\lst@EndWriteFile\egroup

```

Now `\lst@samplebox` contains the verbatim part. If it’s too wide, we use `atop` and `below` instead of `left` and `right`.

```

3771 \ifdim \wd\lst@samplebox>.5\linewidth
3772 \begin{center}%
3773 \hbox to\linewidth{\box\lst@samplebox\hss}%
3774 \end{center}%
3775 \lst@sampleInput

```



```

3776     \else
3777         \begin{center}%
3778         \begin{minipage}{0.45\linewidth}\lst@sampleInput\end{minipage}%
3779         \quad
3780         \begin{minipage}{0.45\linewidth}%
3781             \hbox to\linewidth{\box\lst@samplebox\hss}%
3782         \end{minipage}%
3783         \end{center}%
3784     \fi}

```

The new keyword class point.

```

3785 \lst@InstallKeywords{p}{point}{pointstyle}\relax{keywordstyle}{\ld

```

`\lstxsample` Omitting `\lst@EndWriteFile` leaves the file open.

```

3786 \lstnewenvironment{\lstxsample}[1] []
3787     {\begingroup
3788         \setkeys{lst}{belowskip=-\medskipamount,language={},style={},%
3789             tabsize=4,gobble=5,basicstyle=\small\ttfamily,%
3790             basewidth=0.51em,point={#1}}
3791         \lst@BeginAlsoWriteFile{\jobname.tmp}}
3792     {\endgroup
3793     \endgroup}

```

`\lst@sampleInput` inputs the ‘left-hand’ side.

```

3794 \def\lst@sampleInput{%
3795     \MakePercentComment\catcode'\^M=10\relax
3796     \small\lst@sample
3797     {\setkeys{lst}{SelectCharTable=\lst@ReplaceInput{\^I}%
3798         {\lst@ProcessTabulator}}}%
3799     \leavevmode \input{\jobname.tmp}\MakePercentIgnore}

```

19.5 Miscellaneous

Sectioning and cross referencing We begin with a redefinition paragraph.

```

3800 \renewcommand\paragraph{\@startsection{paragraph}{4}{\z0}%
3801     {1.25ex \@plus1ex \@minus.2ex}%
3802     {-1em}%
3803     {\normalfont\normalsize\bfseries}}

```

We introduce `\lstref` which prints section number together with its name.

```

3804 \def\lstref#1{\emph{\ref{#1} \nameref{#1}}}

```

Moreover we adjust the table of contents.

```

3805 \def\@part[#1]#2{\addcontentsline{toc}{part}{#1}%
3806     {\parindent\z0 \raggedright \interlinepenalty\@M
3807         \normalfont \huge \bfseries #2\markboth{}{}\par}%
3808     \nobreak\vskip 3ex\@afterheading}
3809 \renewcommand*\l@section[2]{%
3810     \addpenalty\@secpenalty
3811     \addvspace{.25em \@plus\p0}%
3812     \setlength\@tempdima{1.5em}%
3813     \begingroup
3814         \parindent \z0 \rightskip \@pnumwidth
3815         \parfillskip -\@pnumwidth
3816         \leavevmode

```

```

3817 \advance\leftskip\@tempdima
3818 \hskip -\leftskip
3819 #1\nobreak\hfil \nobreak\hb@xt@\@pnumwidth{\hss #2}\par
3820 \endgroup}
3821 \renewcommand*\l@section{\@dottedtocline{2}{0pt}{2.3em}}
3822 \renewcommand*\l@subsubsection{\@dottedtocline{3}{0pt}{3.2em}}

```

Indexing The ‘user’ commands. `\rstyle` is defined below.

```

3823 \newcommand\ikeyname[1]{%
3824 \lstkeyindex{#1}{}%
3825 \lstaspectindex{#1}{}%
3826 \keyname{#1}}
3827 \newcommand\ekeyname[1]{%
3828 \@bsphack
3829 \lstkeyindex{#1}{\encapchar usage}%
3830 \lstaspectindex{#1}{\encapchar usage}%
3831 \@esphack}
3832 \newcommand\rkeyname[1]{%
3833 \@bsphack
3834 \lstkeyindex{#1}{\encapchar main}%
3835 \lstaspectindex{#1}{\encapchar main}%
3836 \@esphack{\rstyle\keyname{#1}}}
3837 \newcommand\icmdname[1]{%
3838 \@bsphack
3839 \lstaspectindex{#1}{}%
3840 \@esphack\texttt{\string#1}}
3841 \newcommand\rcmdname[1]{%
3842 \@bsphack
3843 \lstaspectindex{#1}{\encapchar main}%
3844 \@esphack\texttt{\rstyle\string#1}}

```

One of the two yet unknown ‘index’-macros is empty, the other looks up the aspect name for the given argument.

```

3845 \def\lstaspectindex#1#2{%
3846 \global\@namedef{lstkandc@\string#1}{}%
3847 \@ifundefined{lstisaspect@\string#1}
3848 {\index{unknown\levelchar
3849 \protect\texttt{\protect\string\string#1}\#2}}%
3850 {\index{\@nameuse{lstisaspect@\string#1}\levelchar
3851 \protect\texttt{\protect\string\string#1}\#2}}%
3852 }
3853 \def\lstkeyindex#1#2{%
3854 % \index{key\levelchar\protect\keyname{#1}\#2}%
3855 }

```

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.

```

3856 \def\lstisaspect[#1]\#2{%
3857 \global\@namedef{lstaspect@#1}{#2}%
3858 \lst@AddTo\lst@allkeysandcmds{,#2}%
3859 \@for\lst@temp:=#2\do
3860 {\ifx\@empty\lst@temp\else
3861 \global\@namedef{lstisaspect@\lst@temp}{#1}%

```

```

3862     \fi}}
3863 \gdef\lst@allkeysandcmds{}

This relation is also good to print all keys and commands of a particular aspect
...

3864 \def\lstprintaspectkeysandcmds#1{%
3865     \lst@true
3866     \expandafter\@for\expandafter\lst@temp
3867     \expandafter:\expandafter=\csname lstaspect@#1\endcsname\do
3868     {\lst@if\lst@false\else, \fi \texttt{\lst@temp}}}

... or to check the reference. Note that we've defined \lstkandc@⟨name⟩ in
\lstaspectindex.

3869 \def\lstcheckreference{%
3870     \@for\lst@temp:=\lst@allkeysandcmds\do
3871     {\ifx\lst@temp\@empty\else
3872         \@ifundefined{lstkandc@\lst@temp}
3873         {\typeout{\lst@temp\space not in reference guide?}}{\fi}%
3874     \fi}}

```

Unique styles

```

3875 \newcommand*\lst{\texttt{lst}}
3876 \newcommand*\Cpp{C\texttt{++}}
3877 \let\keyname\texttt
3878 \let\keyvalue\texttt
3879 \let\hookname\texttt
3880 \newcommand*\aspectname[1]{\{\normalfont\sffamily#1\}}

3881 \DeclareRobustCommand\packagename[1]{%
3882     {\leavevmode\text@command{#1}%
3883     \switchfontfamily\sfdefault\rmdefault
3884     \check@ic1 #1\check@icr
3885     \expandafter}}%
3886 \renewcommand\packagename[1]{\{\normalfont\sffamily#1\}}
3887 \def\switchfontfamily#1#2{%
3888     \begingroup\xdef\@gtempa{#1}\endgroup
3889     \ifx\f@family\@gtempa\fontfamily#2%
3890         \else\fontfamily#1\fi
3891     \selectfont}

```

The color mainly for keys and commands in the reference guide.

```

3892 \ifcolor
3893     \definecolor{darkgreen}{rgb}{0,0.5,0}
3894     \def\rstyle{\color{darkgreen}}
3895 \else
3896     \let\rstyle\empty
3897 \fi

```

Commands for credits and helpers

```

3898 \gdef\lst@emails{}
3899 \newcommand*\lstthanks[2]
3900     {#1\lst@AddTo\lst@emails{,#1,<#2>}}%
3901     \ifx\@empty#2\@empty\typeout{Missing email for #1}\fi}
3902 \newcommand*\lsthelper[3]
3903     {\let~\ #1}%
3904     \lst@ifOneOf#1\relax\lst@emails

```

```
3905 {}{\typeout{^^JWarning: Unknown helper #1.^^J}}
```

Languages and styles

```
3906 \lstdefinelanguage[doc]{Pascal}{%
3907   morekeywords={alfa,and,array,begin,boolean,byte,case,char,const,div,%
3908     do,downto,else,end,false,file,for,function,get,goto,if,in,%
3909     integer,label,maxint,mod,new,not,of,or,pack,packed,page,program,%
3910     procedure,put,read,readln,real,record,repeat,reset,rewrite,set,%
3911     text,then,to,true,type,unpack,until,var,while,with,write,writeln},%
3912   sensitive=false,%
3913   morecomment=[s]{(*){*)},%
3914   morecomment=[s]{\}{\}},%
3915   morestring=[d]{'}}
3916 \lstdefinestyle{}
3917   {basicstyle={},%
3918     keywordstyle=\bfseries,identifierstyle={},%
3919     commentstyle=\itshape,stringstyle={},%
3920     numberstyle={},stepnumber=1,%
3921     pointstyle=\pointstyle}
3922 \def\pointstyle{%
3923   {\let\lst@um\@empty \xdef\@gtempa{\the\lst@token}}%
3924   \expandafter\lstkeyindex\expandafter{\@gtempa}{}%
3925   \expandafter\lstaspectindex\expandafter{\@gtempa}{}%
3926   \rstyle}
3927 \lstset{defaultdialect=[doc]Pascal,language=Pascal,style={}}
```

19.6 Scanning languages

`\lstscanlanguages` We modify some internal definitions and input the files.

```
3928 \def\lstscanlanguages#1#2#3{%
3929   \begingroup
3930     \def\lst@DefDriver@##1##2##3##4[##5]##6{%
3931       \lst@false
3932       \lst@lAddTo\lst@scan{##6(##5),}%
3933       \begingroup
3934       \ifnextchar[{\lst@XDefDriver{##1}##3}{\lst@DefDriver@##3}}%
3935     \def\lst@XDefDriver[##1]{}%
3936     \lst@InputCatcodes
3937     \def\lst@dontinput{#3}%
3938     \let\lst@scan\@empty
3939     \lst@for{#2}\do{%
3940       \lst@ifOneOf##1\relax\lst@dontinput
3941       {}%
3942       {\InputIfFileExists{##1}{}}}%
3943     \global\let\@gtempa\lst@scan
3944   \endgroup
3945   \let#1\@gtempa}
```

`\lstprintlanguages` `\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.

```
3946 \def\lstprintlanguages#1{%
3947   \def\do##1{\setbox\@tempboxa\hbox{##1\space\space}}%
```

```

3948         \ifdim\wd\@tempboxa<.5\linewidth \wd\@tempboxa.5\linewidth
3949                                     \else \wd\@tempboxa\linewidth \fi
3950     \box\@tempboxa\allowbreak}%
3951 \begin{quote}
3952     \par\noindent
3953     \hyphenpenalty=\@M \rightskip=\z@\@plus\linewidth\relax
3954     \lst@BubbleSort#1%
3955     \expandafter\lst@NextLanguage#1\relax(\relax),%
3956 \end{quote}}

    We get and define the current language and ...
3957 \def\lst@NextLanguage#1(#2),{%
3958     \ifx\relax#1\else
3959         \def\lst@language{#1}\def\lst@dialects{(#2),}%
3960         \expandafter\lst@NextLanguage@
3961     \fi}

    ... gather all available dialect of this language (note that the list has been sorted)
3962 \def\lst@NextLanguage@#1(#2),{%
3963     \def\lst@temp{#1}%
3964     \ifx\lst@temp\lst@language
3965         \lst@lAddTo\lst@dialects{(#2),}%
3966         \expandafter\lst@NextLanguage@
3967     \else
        or begin to print this language with all its dialects. Therefor we sort the dialects
3968         \do{\lst@language
3969             \ifx\lst@dialects\lst@emptydialect\else
3970                 \expandafter\lst@NormedDef\expandafter\lst@language
3971                 \expandafter{\lst@language}%
3972                 \space%
3973                 \lst@BubbleSort\lst@dialects
3974                 \expandafter\lst@PrintDialects\lst@dialects(\relax),%
3975                 )%
3976             \fi}%
3977         \def\lst@next{\lst@NextLanguage#1(#2),}%
3978         \expandafter\lst@next
3979     \fi}
3980 \def\lst@emptydialect{(),}

    and print the dialect with appropriate commas in between.
3981 \def\lst@PrintDialects(#1),{%
3982     \ifx\@empty#1\@empty empty\else
3983         \lst@PrintDialect{#1}%
3984     \fi
3985     \lst@PrintDialects@}
3986 \def\lst@PrintDialects@(#1),{%
3987     \ifx\relax#1\else
3988         , \lst@PrintDialect{#1}%
3989     \expandafter\lst@PrintDialects@
3990 \fi}

    Here we take care of default dialects.
3991 \def\lst@PrintDialect#1{%
3992     \lst@NormedDef\lst@temp{#1}%
3993     \expandafter\ifx\csname\@lst dd@\lst@language\endcsname\lst@temp

```

```

3994      \texttt{\underbar{#1}}%
3995      \else
3996      \texttt{#1}%
3997      \fi}

```

19.7 Bubble sort

`\lst@ifLE` $\langle string\ 1 \rangle$ relax `\@empty` $\langle string\ 2 \rangle$ relax `\@empty` $\langle then \rangle$ $\langle else \rangle$. If $\langle string\ 1 \rangle \leq \langle string\ 2 \rangle$, we execute $\langle then \rangle$ and $\langle else \rangle$ otherwise. Note that this comparison is case insensitive.

```

3998 \def\lst@ifLE#1#2\@empty#3#4\@empty{%
3999     \ifx #1\relax
4000         \let\lst@next\@firstoftwo
4001     \else \ifx #3\relax
4002         \let\lst@next\@secondoftwo
4003     \else
4004         \lowercase{\ifx#1#3}%
4005         \def\lst@next{\lst@ifLE#2\@empty#4\@empty}%
4006     \else
4007         \lowercase{\ifnum'#1<'#3}\relax
4008         \let\lst@next\@firstoftwo
4009     \else
4010         \let\lst@next\@secondoftwo
4011     \fi
4012 \fi
4013 \fi \fi
4014 \lst@next}

```

`\lst@BubbleSort` is in fact a derivation of bubble sort.

```

4015 \def\lst@BubbleSort#1{%
4016     \ifx\@empty#1\else
4017         \lst@false
         We ‘bubble sort’ the first, second, ... elements and ...
4018         \expandafter\lst@BubbleSort@#1\relax,\relax,%
         ... then the second, third, ... elements until no elements have been swapped.
4019         \expandafter\lst@BubbleSort@\expandafter,\lst@sorted
4020                                     \relax,\relax,%
4021         \let#1\lst@sorted
4022         \lst@if
4023             \def\lst@next{\lst@BubbleSort#1}%
4024             \expandafter\expandafter\expandafter\lst@next
4025         \fi
4026     \fi}
4027 \def\lst@BubbleSort@#1,#2,{%
4028     \ifx\@empty#1\@empty
4029         \def\lst@sorted{#2}%
4030         \def\lst@next{\lst@BubbleSort@@}%
4031     \else
4032         \let\lst@sorted\@empty
4033         \def\lst@next{\lst@BubbleSort@@#1,#2,%}
4034     \fi
4035     \lst@next}

```

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).

```

4036 \def\lst@BubbleSort@@#1,#2,{%
4037   \ifx\relax#1\else
4038     \ifx\relax#2%
4039       \lst@lAddTo\lst@sorted{#1,}%
4040       \expandafter\expandafter\expandafter\lst@BubbleSort@@@
4041     \else
4042       \lst@ifLE #1\relax\@empty #2\relax\@empty
4043         {\lst@lAddTo\lst@sorted{#1,#2,}}%
4044         {\lst@true \lst@lAddTo\lst@sorted{#2,#1,}}%
4045       \expandafter\expandafter\expandafter\lst@BubbleSort@@
4046     \fi
4047   \fi}
4048 \def\lst@BubbleSort@@@#1\relax,{%
4049 </doc>

```

20 Interfaces to other programs

20.1 0.21 compatibility

Some keys have just been renamed.

```

4050 <{*0.21}>
4051 \lst@BeginAspect{0.21}
4052 \lst@Key{labelstyle}{-}{\def\lst@numberstyle{#1}}
4053 \lst@Key{labelsep}{10pt}{\def\lst@numbersep{#1}}
4054 \lst@Key{labelstep}{0}{%
4055   \ifnum #1=\z@ \KV@lst@numbers{none}%
4056   \else \KV@lst@numbers{left}\fi
4057   \def\lst@stepnumber{#1\relax}}
4058 \lst@Key{firstlabel}\relax{\def\lst@firstnumber{#1\relax}}
4059 \lst@Key{advancelabel}\relax{\def\lst@advancenumber{#1\relax}}
4060 \let\c@lstlabel\c@lstnumber
4061 \lst@AddToHook{Init}{\def\thelstnumber{\thelstlabel}}
4062 \newcommand*\thelstlabel{\@arabic\c@lstlabel}

A \let in the second last line has been changed to \def after a bug report by
Venkatesh Prasad Ranganath.

4063 \lst@Key{first}\relax{\def\lst@firstline{#1\relax}}
4064 \lst@Key{last}\relax{\def\lst@lastline{#1\relax}}

4065 \lst@Key{framerulewidth}{.4pt}{\def\lst@framerulewidth{#1}}
4066 \lst@Key{framerulesep}{2pt}{\def\lst@rulesep{#1}}
4067 \lst@Key{frametextsep}{3pt}{\def\lst@frametextsep{#1}}
4068 \lst@Key{framerulecolor}{-}{\lstKV@OptArg[]{#1}%
4069   {\ifx\@empty##2\@empty
4070     \let\lst@rulecolor\@empty
4071   \else
4072     \ifx\@empty##1\@empty
4073       \def\lst@rulecolor{\color{##2}}%
4074     \else

```

```

4075         \def\lst@rulecolor{\color{##1}{##2}}%
4076     \fi
4077 \fi}}
4078 \lst@Key{backgroundcolor}{\z@}{\lstKV@OptArg[]{#1}%
4079     {\ifx\@empty##2\@empty
4080         \let\lst@bkgcolor\@empty
4081     \else
4082         \ifx\@empty##1\@empty
4083             \def\lst@bkgcolor{\color{##2}}%
4084         \else
4085             \def\lst@bkgcolor{\color{##1}{##2}}%
4086         \fi
4087     \fi}}
4088 \lst@Key{framespread}{\z@}{\def\lst@framespread{#1}}
4089 \lst@AddToHook{PreInit}
4090     {\@tempdima\lst@framespread\relax \divide\@tempdima\tw@
4091     \edef\lst@framextopmargin{\the\@tempdima}%
4092     \let\lst@framexrightmargin\lst@framextopmargin
4093     \let\lst@framexbottommargin\lst@framextopmargin
4094     \advance\@tempdima\lst@xleftmargin\relax
4095     \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.

```

4096 \newdimen\lst@innerspread \newdimen\lst@outerspread
4097 \lst@Key{spread}{\z@,\z@}{\lstKV@CSTwoArg{#1}%
4098     {\lst@innerspread##1\relax
4099     \ifx\@empty##2\@empty
4100         \divide\lst@innerspread\tw@\relax
4101         \lst@outerspread\lst@innerspread
4102     \else
4103         \lst@outerspread##2\relax
4104     \fi}}
4105 \lst@AddToHook{BoxUnsafe}{\lst@outerspread\z@ \lst@innerspread\z@}
4106 \lst@Key{wholeline}{false}[t]{\lstKV@SetIf{#1}\lst@ifresetmargins}
4107 \lst@Key{indent}{\z@}{\def\lst@xleftmargin{#1}}
4108 \lst@AddToHook{PreInit}
4109     {\lst@innerspread=-\lst@innerspread
4110     \lst@outerspread=-\lst@outerspread
4111     \ifodd\c@page \advance\lst@innerspread\lst@xleftmargin
4112     \else \advance\lst@outerspread\lst@xleftmargin \fi
4113     \ifodd\c@page
4114         \edef\lst@xleftmargin{\the\lst@innerspread}%
4115         \edef\lst@xrightmargin{\the\lst@outerspread}%
4116     \else
4117         \edef\lst@xleftmargin{\the\lst@outerspread}%
4118         \edef\lst@xrightmargin{\the\lst@innerspread}%
4119     \fi}
4120 \lst@Key{defaultclass}\relax{\def\lst@classoffset{#1}}
4121 \lst@Key{stringtest}\relax{}% dummy
4122 \lst@Key{outputpos}\relax{\lst@outputpos#1\relax\relax}
4123 \lst@Key{stringspaces}\relax[t]{\lstKV@SetIf{#1}\lst@ifshowstringspaces}
4124 \lst@Key{visiblespaces}\relax[t]{\lstKV@SetIf{#1}\lst@ifshowspaces}
4125 \lst@Key{visibletabs}\relax[t]{\lstKV@SetIf{#1}\lst@ifshowtabs}

```



```

4126 \lst@EndAspect
4127 </0.21>

```

20.2 fancyvrb

Denis Girou asked whether fancyvrb and listings could work together.

fancyvrb We set the boolean and call a submacro.

```

4128 <*kernel>
4129 \lst@Key{fancyvrb}\relax[t]{%
4130     \lstKV@SetIf{#1}\lst@iffancyvrb
4131     \lstFV@fancyvrb}
4132 \ifx\lstFV@fancyvrb\@undefined
4133     \gdef\lstFV@fancyvrb{\lst@RequireAspects{fancyvrb}\lstFV@fancyvrb}
4134 \fi
4135 </kernel>

```

We end the job if fancyvrb is not present.

```

4136 <*misc>
4137 \lst@BeginAspect{fancyvrb}
4138 \@ifundefined{FancyVerbFormatLine}
4139     {\typeout{^^J%
4140         ***^^J%
4141         *** ‘listings.sty’ needs ‘fancyvrb.sty’ right now.^^J%
4142         *** Please ensure its availability and try again.^^J%
4143         ***^^J}%
4144     \batchmode \@@end}{-}

```

\lstFV@fancyvrb We assign the correct \FancyVerbFormatLine macro.

```

4145 \gdef\lstFV@fancyvrb{%
4146     \lst@iffancyvrb
4147         \ifx\FancyVerbFormatLine\lstFV@FancyVerbFormatLine\else
4148             \let\lstFV@FVFL\FancyVerbFormatLine
4149             \let\FancyVerbFormatLine\lstFV@FancyVerbFormatLine
4150         \fi
4151     \else
4152         \ifx\lstFV@FVFL\@undefined\else
4153             \let\FancyVerbFormatLine\lstFV@FVFL
4154             \let\lstFV@FVFL\@undefined
4155         \fi
4156     \fi}

```

\lstFV@VerbatimBegin We initialize things if necessary.

```

4157 \gdef\lstFV@VerbatimBegin{%
4158     \ifx\FancyVerbFormatLine\lstFV@FancyVerbFormatLine
4159         \lsthk@TextStyle \lsthk@BoxUnsafe
4160         \lsthk@PreSet
4161         \lst@activecharsfalse
4162         \let\normalbaselines\relax

```

To do: Is this \let bad?

I inserted `\lst@ifresetmargins... \fi` after a bug report from Peter Bartke. The linewidth is saved and restored since a bug report by Denis Girou.

```
4163 \xdef\lstFV@RestoreData{\noexpand\linewidth\the\linewidth\relax}%
4164     \lst@Init\relax
4165     \lst@ifresetmargins \advance\linewidth-\@totalleftmargin \fi
4166 \lstFV@RestoreData
4167     \everypar{}\global\lst@newlines\z@
4168     \lst@mode\lst@nomode \let\lst@entermodes\@empty
4169     \lst@InterruptModes
```

Rolf Niepraschk reported a bug concerning ligatures to Denis Girou.

```
4170 %% D.G. modification begin - Nov. 25, 1998
4171     \let\@noligs\relax
4172 %% D.G. modification end
4173     \fi}
```

`\lstFV@VerbatimEnd` A box and macro must exist after `\lst@DeInit`. We store them globally.

```
4174 \gdef\lstFV@VerbatimEnd{%
4175     \ifx\FancyVerbFormatLine\lstFV@FancyVerbFormatLine
4176         \global\setbox\lstFV@gtempboxa\box\@tempboxa
4177         \global\let\@gtempa\FV@ProcessLine
4178         \lst@mode\lst@Pmode
4179         \lst@DeInit
4180         \let\FV@ProcessLine\@gtempa
4181         \setbox\@tempboxa\box\lstFV@gtempboxa
4182         \par
4183     \fi}
```

The `\par` has been added after a bug report by Peter Bartke.

```
4184 \newbox\lstFV@gtempboxa
```

We insert `\lstFV@VerbatimBegin` and `\lstFV@VerbatimEnd` where necessary.

```
4185 \lst@AddTo\FV@VerbatimBegin\lstFV@VerbatimBegin
4186 \lst@AddToAtTop\FV@VerbatimEnd\lstFV@VerbatimEnd
4187 \lst@AddTo\FV@LVerbatimBegin\lstFV@VerbatimBegin
4188 \lst@AddToAtTop\FV@LVerbatimEnd\lstFV@VerbatimEnd
4189 \lst@AddTo\FV@BVerbatimBegin\lstFV@VerbatimBegin
4190 \lst@AddToAtTop\FV@BVerbatimEnd\lstFV@VerbatimEnd
```

`\lstFV@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.

```
4191 \gdef\lstFV@FancyVerbFormatLine#1{%
4192     \let\lst@arg\@empty \lst@FVConvert#1\@nil
4193     \global\lst@newlines\z@
4194     \vtop{\noindent\lst@parshape
4195         \lst@ReenterModes
4196         \lst@arg \lst@PrintToken\lst@EOLUpdate\lsthk@InitVarsBOL
4197         \lst@InterruptModes}}
```

The `\lst@parshape` inside `\vtop` is due to a bug report from Peter Bartke. A `\leavevmode` became `\noindent`.

`fvcmdparams` These keys adjust `\lst@FVcmdparams`, which will be used by the following `morefvcmdparams` version macro. The base set of commands and parameter numbers was provided by Denis Girou.

```
4198 \lst@Key{fvcmdparams}%
4199     {\overlay\@ne}%
4200     {\def\lst@FVcmdparams{, #1}}
4201 \lst@Key{morefvcmdparams}\relax{\lst@lAddTo\lst@FVcmdparams{, #1}}
```

`\lst@FVConvert` We do conversion or ...

```
4202 \gdef\lst@FVConvert{\@tempcnta\z@ \lst@FVConvert00}%
4203 \gdef\lst@FVConvert00{%
4204     \ifcase\@tempcnta
4205         \expandafter\futurelet\expandafter\@let@token
4206         \expandafter\lst@FVConvert00
4207     \else
```

... we append arguments without conversion, argument by argument, `\@tempcnta` times.

```
4208         \expandafter\lst@FVConvert00a
4209     \fi}
4210 \gdef\lst@FVConvert00a#1{%
4211     \lst@lAddTo\lst@arg{\#1}\advance\@tempcnta\m@ne
4212     \lst@FVConvert00}%

```

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.

```
4213 \gdef\lst@FVConvert00{%
4214     \ifcat\noexpand\@let@token\bgroup \expandafter\lst@FVConvertArg
4215                                     \else \expandafter\lst@FVConvert0 \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.

```
4216 \gdef\lst@FVConvertArg#1{%
4217     {\let\lst@arg\@empty
4218     \lst@FVConvert#1\@nil
4219     \global\let\@gtempa\lst@arg}%
4220     \lst@lExtend\lst@arg{\expandafter{\@gtempa\lst@PrintToken}}}%
4221     \lst@FVConvert}

4222 \gdef\lst@FVConvert@#1{%
4223     \ifx \@nil#1\else
4224         \if\relax\noexpand#1%
4225             \lst@lAddTo\lst@arg{\lst@OutputLostSpace\lst@PrintToken#1}%
4226         \else
4227             \lccode'\~='#1\lowercase{\lst@lAddTo\lst@arg~}%
4228         \fi
4229         \expandafter\lst@FVConvert
4230     \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).

```
4231 \gdef\lst@FVConvert@#1{%
4232     \ifx \@nil#1\else

```

```

4233     \if\relax\noexpand#1%
4234     \lst@lAddTo\lst@arg{\lst@OutputLostSpace\lst@PrintToken#1}%
Here we check for registered commands with arguments and set the value of
\@tempcnta as required.
4235     \def\lst@temp##1,#1##2,##3##4\relax{%
4236         \ifx##3\@empty \else \@tempcnta##2\relax \fi}%
4237     \expandafter\lst@temp\lst@FVcmdparams,#1\z@,\@empty\relax
4238     \else
4239     \lccode'\~='#1\lowercase{\lst@lAddTo\lst@arg~}%
4240     \fi
4241     \expandafter\lst@FVConvert0@
4242     \fi}

4243 \lst@EndAspect
4244 </misc>

```

20.3 Omega support

Ω support looks easy—I hope it works at least in some cases.

```

4245 <*kernel>
4246 \@ifundefined{ocp}{%
4247     {\lst@AddToHook{OutputBox}%
4248         {\let\lst@ProcessLetter\@firstofone
4249           \let\lst@ProcessDigit\@firstofone
4250           \let\lst@ProcessOther\@firstofone}}
4251 </kernel>

```

20.4 LGrind

is used to extract the language names from \lst@arg (the LGrind definition).

```

\lst@LGGetNames 4252 <*misc>
4253 \lst@BeginAspect[keywords,comments,strings,language]{lgrind}
4254 \gdef\lst@LGGetNames#1:#2\relax{%
4255     \lst@NormedDef\lstlang@{#1}\lst@ReplaceInArg\lstlang@{|,}%
4256     \def\lst@arg{: #2}}

```

\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

```

4257 \gdef\lst@LGGetValue#1{%
4258     \lst@false
4259     \def\lst@temp##1:#1##2##3\relax{%
4260         \ifx\@empty##2\else \lst@LGGetValue@{#1}\fi}
4261     \expandafter\lst@temp\lst@arg:#1\@empty\relax}
remove the pair if necessary.
4262 \gdef\lst@LGGetValue@#1{%
4263     \lst@true
4264     \def\lst@temp##1:#1##2:##3\relax{%
4265         \@ifnextchar=\lst@LGGetValue@@{\lst@LGGetValue@@=##2\relax
4266         \def\lst@arg{##1:##3}}}%
4267     \expandafter\lst@temp\lst@arg\relax}
4268 \gdef\lst@LGGetValue@@=#1\relax{\def\lst@LGvalue{#1}}

```

`\lst@LGGetComment` 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.

```

4269 \gdef\lst@LGGetComment#1#2{%
4270     \let#2\@empty
4271     \lst@LGGetValue{#1b}%
4272     \lst@if
4273         \let#2\lst@LGvalue
4274         \lst@LGGetValue{#1e}%
4275         \ifx\lst@LGvalue\lst@LGEOL
4276             \edef\lstlang@{\lstlang@,commentline={#2}}%
4277             \let#2\@empty
4278         \else
4279             \edef#2{{#2}{\lst@LGvalue}}%
4280         \fi
4281     \fi}

```

`\lst@LGGetString` does the same for string delimiters, but it doesn't 'return' any value.

```

4282 \gdef\lst@LGGetString#1#2{%
4283     \lst@LGGetValue{#1b}%
4284     \lst@if
4285         \let#2\lst@LGvalue
4286         \lst@LGGetValue{#1e}%
4287         \ifx\lst@LGvalue\lst@LGEOL
4288             \edef\lstlang@{\lstlang@,morestringizer=[l]{#2}}%
4289         \else

```

we must check for \e, i.e. whether we have to use doubled or backslashed stringizer.

```

4290         \ifx #2\lst@LGvalue
4291             \edef\lstlang@{\lstlang@,morestringizer=[d]{#2}}%
4292         \else
4293             \edef\lst@temp{\lst@LGe#2}%
4294             \ifx \lst@temp\lst@LGvalue
4295                 \edef\lstlang@{\lstlang@,morestringizer=[b]{#2}}%
4296             \else
4297                 \PackageWarning{Listings}%
4298                 {String #2...\lst@LGvalue\space not supported}%
4299             \fi
4300         \fi
4301     \fi
4302 \fi}

```

`\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 :.`

```

4303 \gdef\lst@LGDefLang{%
4304     \lst@LGReplace
4305     \let\lstlang@\empty

```

Get the keywords and values of friends.

```

4306     \lst@LGGetValue{kw}%
4307     \lst@if
4308         \lst@ReplaceInArg\lst@LGvalue{{ }},}%
4309         \edef\lstlang@{\lstlang@,keywords={\lst@LGvalue}}%
4310     \fi

```

```

4311 \lst@LGGetValue{oc}%
4312 \lst@if
4313 \edef\lstlang@{\lstlang@,sensitive=f}%
4314 \fi
4315 \lst@LGGetValue{id}%
4316 \lst@if
4317 \edef\lstlang@{\lstlang@,alsoletter=\lst@LGvalue}%
4318 \fi

```

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.

```

4319 \lst@LGGetComment a\lst@LGa
4320 \lst@LGGetComment c\lst@LGc
4321 \ifx\lst@LGa\@empty
4322 \ifx\lst@LGc\@empty\else
4323 \edef\lstlang@{\lstlang@,singlecomment=\lst@LGc}%
4324 \fi
4325 \else
4326 \ifx\lst@LGc\@empty
4327 \edef\lstlang@{\lstlang@,singlecomment=\lst@LGa}%
4328 \else
4329 \edef\lstlang@{\lstlang@,doublecomment=\lst@LGc\lst@LGa}%
4330 \fi
4331 \fi

```

Now we parse the stringizers.

```

4332 \lst@LGGetString s\lst@LGa
4333 \lst@LGGetString l\lst@LGa

```

We test for the continuation capability and

```

4334 \lst@LGGetValue{tc}%
4335 \lst@if
4336 \edef\lstlang@{\lstlang@,lgrindef=\lst@LGvalue}%
4337 \fi

```

define the language.

```

4338 \expandafter\xdef\csname\@lst LGl@ng@\lst@language@\endcsname
4339 {\noexpand\lstset{\lstlang@}}%

```

Finally we inform the user of all ignored capabilities.

```

4340 \lst@ReplaceInArg\lst@arg{{: :}:}\let\lst@LGvalue\@empty
4341 \expandafter\lst@LGDroppedCaps\lst@arg\relax\relax
4342 \ifx\lst@LGvalue\@empty\else
4343 \PackageWarningNoLine{Listings}{Ignored capabilities for
4344 \space '\lst@language@' are\MessageBreak\lst@LGvalue}%
4345 \fi}

```

`\lst@LGDroppedCaps` just drops a previous value and appends the next capability name to `\lst@LGvalue`.

```

4346 \gdef\lst@LGDroppedCaps#1:#2#3{%
4347 \ifx#2\relax
4348 \lst@RemoveCommas\lst@LGvalue
4349 \else
4350 \edef\lst@LGvalue{\lst@LGvalue,#2#3}%
4351 \expandafter\lst@LGDroppedCaps
4352 \fi}

```

<code>\lst@LGReplace</code>	We replace ‘escaped : ~\$’ by catcode 11 versions, and other strings by some kind
<code>\lst@LGe</code>	of short versions (which is necessary to get the above definitions work).

```

4353 \begingroup
4354 \catcode'\/=0
4355 \lccode'\z=':\lccode'\y='^\lccode'\x='$\lccode'\v='|
4356 \catcode'\=12\relax
4357 /lowercase{%
4358 /gdef/1st@LGRReplace{/1st@ReplaceInArg/1st@arg
4359     {\{:}\{z \}{\~}\{y}{\${x}\{|\}{v}{ \}{ \}{:}\{ \}{ \}{\{(\{ \})}}}}
4360 /gdef/1st@LGe{\e}
4361 }
4362 /endgroup

```

`\lst@LGRead` reads one language definition and defines the language if the correct one is found.

```

4363 \gdef\lst@LGRead#1\par{%
4364   \lst@LGGetNames#1:\relax
4365   \def\lst@temp{endoflanguagedefinitions}%
4366   \ifx\lstlang@\lst@temp
4367     \let\lst@next\endinput
4368   \else
4369     \expandafter\lst@IfOneOf\lst@language@\relax\lstlang@
4370     {\lst@LGDefLang \let\lst@next\endinput}%
4371     {\let\lst@next\lst@LGRead}%
4372   \fi
4373   \lst@next}

```

`\lgrindef` We only have to request the language and

```

4374 \lst@Key{lgrindf}\relax{%
4375   \lst@NormedDef\lst@language@{#1}%
4376   \begingroup
4377   \@ifundefined{lstLGleng@\lst@language@}%
4378     {\everypar{\lst@LGRead}%
4379      \catcode'\=12\catcode'\{=12\catcode'\}=12\catcode'\%=12%
4380      \catcode'\#=14\catcode'\$=12\catcode'\^=12\catcode'\_ =12\relax
4381      \input{\lstlgrindfile}%
4382     }{%
4383   \endgroup

```

select it or issue an error message.

```

4384   \@ifundefined{lstLGleng@\lst@language@}%
4385     {\PackageError{Listings}%
4386      {LGrind language \lst@language@\space undefined}%
4387      {The language is not loadable. \@ehc}}%
4388     {\lsthk@SetLanguage
4389      \csname@\lst LGleng@\lst@language@\endcsname}}

```

`\lstlgrindefile` contains just the file name.

```

4390 \@ifundefined{lstlgrindefile}
4391     {\lst@UserCommand\lstlgrindefile{lggrindef.}}{}
4392 \lst@EndAspect
4393 \/\misc

```

20.5 hyperref

```
4394 (*misc)
4395 \lst@BeginAspect[keywords]{hyper}
```

hyperanchor determine the macro to set an anchor and a link, respectively.

```
hyperlink 4396 \lst@Key{hyperanchor}\hyper@@anchor{\let\lst@hyperanchor#1}
4397 \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 `lstHR@` and a number as suffix). Then we make this free macro equivalent to `\@empty`, so it is not used the next time.

```
4398 \lst@InstallKeywords{h}{hyperref}{\relax}
4399   {\begingroup
4400     \let\lst@UM\@empty \xdef\@gtempa{\the\lst@token}%
4401   \endgroup
4402   \lst@GetFreeMacro{lstHR@\@gtempa}%
4403   \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 ...

```
4404   \@tempcntb\@tempcnta \advance\@tempcntb\m@ne
4405   \edef\lst@alloverstyle##1{%
4406     \let\noexpand\lst@alloverstyle\noexpand\@empty
4407     \noexpand\smash{\raise\baselineskip\hbox
4408       {\noexpand\lst@hyperanchor{lst.\@gtempa\the\@tempcnta}%
4409        {\relax}}}%
... and a link to the last occurrence (if there is any).
4410     \ifnum\@tempcnta=z@ ##1\else
4411       \noexpand\lst@hyperlink{lst.\@gtempa\the\@tempcntb}{##1}%
4412     \fi}%
4413   }
4414   od
4415 \lst@EndAspect
4416 (/misc)
```

21 Epilogue

```
4417 (*kernel)
```

Each option adds the aspect name to `\lst@loadaspects` or removes it from that data macro.

```
4418 \DeclareOption*{\expandafter\lst@ProcessOption\CurrentOption\relax}
4419 \def\lst@ProcessOption#1#2\relax{%
4420   \ifx #1!%
4421     \lst@DeleteKeysIn\lst@loadaspects{#2}%
4422   \else
4423     \lst@lAddTo\lst@loadaspects{,#1#2}%
4424   \fi}
```


The following aspects are loaded by default.

```

4425 \@ifundefined{lst@loadaspects}
4426   {\def\lst@loadaspects{strings,comments,escape,style,language,%
4427     keywords,labels,lineshape,frames,emph,index}%
4428   }{}

We load the patch file, ...
4429 \InputIfFileExists{lstpatch.sty}{}{}

... process the options, ...
4430 \let\lst@ifsavemem\iffalse
4431 \DeclareOption{savemem}{\let\lst@ifsavemem\iftrue}
4432 \DeclareOption{noaspects}{\let\lst@loadaspects\@empty}
4433 \ProcessOptions

... and load the aspects.
4434 \lst@RequireAspects\lst@loadaspects
4435 \let\lst@loadaspects\@empty

If present we select the empty style and language.
4436 \lst@UseHook{SetStyle}\lst@UseHook{EmptyStyle}
4437 \lst@UseHook{SetLanguage}\lst@UseHook{EmptyLanguage}

Finally we load the configuration files.
4438 \InputIfFileExists{listings.cfg}{}{}
4439 \InputIfFileExists{lstlocal.cfg}{}{}
4440 <info>\lst@ReportAllocs
4441 </kernel>

```

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 `\long` except one removed, faster `\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 `lst`-aspects seem to be good
 - user interface uses `keyval` package
 - `fancyvrb` support
- 0.20 from 1999/07/12
 - new keyword detection mechanism
 - new aspects: `writefile`, `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
- 1.0 β 2001/09/21
 - key names synchronized with `fancyvrb`
 - `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

Index

<code>root</code> , 19	<code>escape>escapechar</code> , 38, 53,
<code>square</code> , 19	54
<code>comments>commentstyle</code> , 6, 22	<code>escape>escapeinside</code> , 53, 54
<code>comments>deletecomment</code> , 23	<code>escape>mathescape</code> , 53
<code>comments>morecomment</code> , 21	<code>escape>texcl</code> , 38, 53
<code>emph>emphstyle</code> , 18, 19	<code>formats>\lstdefineformat</code> , 49
<code>emph>emph</code> , 18, 19	<code>formats>format</code> , 49
	<code>frames>backgroundcolor</code> , 18

```

frames>framaround, 16, 17
frames>frame, 16, 35

index>\lstindexmacro, 36
index>indexstyle, 19
index>index, 19

kernel>\lstinline, 12
kernel>\lstinputlisting, 5
kernel>\lstlistingname, 45
kernel>\lstlistlistingname, 45
kernel>\lstlistoflistings,
    17
kernel>\lstset, 12
kernel>aboveskip, 16
kernel>alsodigit, 41
kernel>alsoletter, 41
kernel>basewidth, 37, 40
kernel>basicstyle, 6
kernel>belowskip, 16
kernel>caption, 7, 17
kernel>columns, 20
kernel>extendedchars, 15, 51
kernel>firstline, 5, 12
kernel>formfeed, 14
kernel>gobble, 27, 52, 53
kernel>identifierstyle, 6
kernel>label, 17
kernel>lastline, 12
kernel>lstlisting, 5
kernel>moredelim, 23
kernel>name, 15
kernel>nolol, 17
kernel>showlines, 5
kernel>showspaces, 14

kernel>showtabs, 14
kernel>tabsize, 14, 27
kernel>tab, 14
kernel>title, 17
keywords>classoffset, 28
keywords>keywordstyle, 6
keywords>morekeywords, 21
keywords>sensitive, 21

labels>firstnumber, 15, 16
labels>numbersep, 7, 15
labels>numberstyle, 7, 15
labels>numbers, 7, 15
labels>stepnumber, 7, 15, 16
definelanguage, 40
language>\lstalias, 45
language>\lstloadlanguages, 12
language>alsolanguage, 12
language>defaultdialect, 45
language>language, 12
lineshape>breakindent, 34

strings>deletestring, 23
strings>morestring, 21
strings>showstringspaces, 6
strings>stringstyle, 6
style>style, 21

unknown>b, 22
unknown>d, 22
unknown>is, 23
unknown>l, 22
unknown>n, 22
unknown>s, 22

```