The Listings Package

Maintainer: Jobst Hoffmann[*] <j.hoffmann(at)fh-aachen.de>

2023/02/27 Version 1.9

Abstract

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

---

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

Preface

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

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

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

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

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

User's guide

1 Getting started

1.1 A minimal file

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

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

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

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

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

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

1.2 Typesetting listings

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

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

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

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

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

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

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

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

It can’t be easier.

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

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

```
1 begin
2     { do nothing }
3 end;
```

```
 1 \begin{lstlisting}[firstline=2,
 2                    lastline=5]
 3 for i:=maxint to 0 do
 4 begin
 5     { do nothing }
 6 end;
 7
 8 Write(’Case insensitive ’);
 9 WritE(’Pascal keywords.’);
10 \end{lstlisting}
```

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

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

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

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

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

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

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

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

1.3 Figure out the appearance

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

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

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

end;

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

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

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

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

The styles use two different kinds of commands. `\ttfamily` and `\bfseries` both take no arguments but `\underbar` does; it underlines the following argument. In general, the *very last* command may 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!

1.4 Seduce to use

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

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

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

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

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

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

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

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

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

Floating listings Displayed listings may float:

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

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

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

- → LaTeX's float mechanism allows one to determine the placement of floats. How can I do that with these? You can write ‘`float=tp`’, for example.

Listing 1: A floating example

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

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

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

```
if (i≤0) then i ← 1;
if (i≥0) then i ← 0;
if (i≠0) then i ← 0;
```

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

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

1.5 Alternatives

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

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

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

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

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

LGrind not installed.

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

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

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

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

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

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

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

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

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

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

```
if $i \leq 0$ then                %\begin{algorithmic}
  $i \leftarrow 1$                %\IF{$i\leq0$}
else                              %\STATE $i\gets1$
  if $i \geq 0$ then              %\ELSE\IF{$i\geq0$}
    $i \leftarrow 0$              %\STATE $i\gets0$
  end if                          %\ENDIF\ENDIF
end if                            %\end{algorithmic}
```

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

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

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

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

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

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

2 The next steps

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

2.1 Software license

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

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

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

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

2.2 Package loading

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

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

`0.21`

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

`draft`

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

`final`

Overwrites a global `draft` option.

`savemem`

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

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

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

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

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

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

2.3 The key=value interface

This package uses the keyval package from the graphics bundle by David Carlisle. Each parameter is controlled by an associated key and a user supplied value. For example, `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 6: `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 only used on individual listings. However, your command is not illegal—it has no effect. You have to use these keys inside the optional argument of the environment or input command.

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

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

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

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

2.4 Programming languages

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

Table 1 shows all predefined languages and dialects. Use the listed names as ⟨*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 4.3.4 for '`defaultdialect`'.

→ I have C code mixed with assembler lines. Can listings pretty-print such source code, i.e. highlight keywords and comments of both languages? '`alsolanguage=[`⟨*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 4.3.4.

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

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

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

ABAP (R/2 4.3, R/2 5.0, R/3 3.1, R/3 4.6C, R/3 6.10)
ACM
ACMscript
ACSL
Ada (2005, 83, 95)
Algol (60, 68)
Ant
Assembler (Motorola68k, x86masm)
Awk (gnu, POSIX)
bash
Basic (Visual)
C (ANSI, Handel, Objective, Sharp)
C++ (11, ANSI, GNU, ISO, Visual)
Caml (light, Objective)
CIL
Clean
Cobol (1974, 1985, ibm)
Comal 80
command.com (WinXP)
Comsol
csh
Delphi
Eiffel
Elan
elisp
erlang
Euphoria
Fortran (03, 08, 18, 77, 90, 95)
GAP
GCL
Gnuplot
Go
hansl
Haskell
HTML
IDL (empty, CORBA)
inform
Java (empty, AspectJ)
JVMIS
ksh
Lingo
Lisp (empty, Auto)
LLVM
Logo
Lua (5.0, 5.1, 5.2, 5.3)
make (empty, gnu)
Mathematica (1.0, 11.0, 3.0, 5.2)
Matlab (empty, 5.1)
Mercury
MetaPost
Miranda
Mizar
ML
Modula-2
MuPAD
NASTRAN
Oberon-2
OCL (decorative, OMG)
Octave
OORexx
Oz
Pascal (Borland6, Standard, XSC)
Perl
PHP
PL/I
Plasm
PostScript
POV
Prolog
Promela
PSTricks
Python
R
Reduce
Rexx (empty, VM/XA)
RSL
Ruby
S (empty, PLUS)
SAS
Scala
Scilab
sh
SHELXL
Simula (67, CII, DEC, IBM)
SPARQL
SQL
Swift
tcl (empty, tk)
TeX (AlLaTeX, common, LaTeX, plain, primitive)
VBScript
Verilog
VHDL (empty, AMS)
VRML (97)
XML
XSLT

There is at least one language (VDM, Vienna Development Language, https://en.wikipedia.org/wiki/Vienna_Development_Method[1]) which is not directly supported by the **listings** package. It needs a package for its own: **vdmlisting**. On the other hand **vdmlisting** uses the **listings** package and so it should be mentioned in this context.

2.4.1 Preferences

Sometimes authors of language support provide their own configuration preferences. These may come either from their personal experience or from the settings in an IDE and can be defined as a **listings** style. From version 1.5b of the **listings** package on these styles are provided as files with the name `listings-`⟨*language*⟩`.prf`, ⟨*language*⟩ is the name of the supported programming language in lowercase letters.

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

```
\input{listings-python.prf}
```

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

At the moment there are five such preferences files:

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

All contributors are invited to supply more personal preferences.

2.5 Special characters

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

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

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

---

[1] Vladimir Nikishkin informed that the previous mentioned URL http://www.vdmportal.org doesn't point to a VDM specific page anymore.

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

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

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

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

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

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

→ To make the LaTeX code more readable, I indent the environments’ program listings. How can I remove that indention in the output? Read ‘How to gobble characters’ in section 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 listings from handling extended characters in listings. If you use them, you should load fontenc, inputenc and/or any other package which defines the characters.

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

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

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

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

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

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

2.6 Line numbers

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

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

And we continue the listing:

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

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

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

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

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

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

And we continue the listing:

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

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

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

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

You can also select the lines to be printed, the options ‘`linerange`’ and ‘`consecutivenumbers`’ are your friend. In a presentation for example you don’t need comments for your programs, so you prefer the line numbers being consecutively numbered, but the results should reflect the behaviour of the program—you omit parts of the lengthy output. So you may have the following program and its results.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main(int argc,char* argv↩
                        ↪[]){
 5   int i;
 6   int limit;
 7   if ( argc > 1 ) {
 8     limit = atoi(argv[1]);
 9   } else {
10     limit = 100;
11   }
12   for (i = 1;i <= limit;i++)↩
                        ↪ {
13     printf("Line no. %3.0d\n↩
                        ↪", i)↩
                        ↪;
14   }
15   return 0;
16 }
```

And these are the results:

```
1 Line no.    1
2 Line no.    2
6 Line no.    6
7 Line no.    7
```

```
\begin{lstlisting}[name=Test,
  language={[ansi]C},
  linerange={1-4,6-7,10-14,
    17-19,21-22},
  firstnumber=1]
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char* argv[]){
  /* declaring variables */
  int i;
  int limit;

  /* checking arguments */
  if ( argc > 1 ) {
    limit = atoi(argv[1]);
  } else {
    limit = 100;
  }

  /* counting lines */
  for (i = 1;i <= limit;i++) {
    printf("Line no. %3.0d\n", i);
  }

  return 0;
}

\end{lstlisting}
And these are the results:
\begin{lstlisting}[language={},
  linerange={1-2,6-7},
  consecutivenumbers=false]
Line no.    1
Line no.    2
Line no.    3
Line no.    4
Line no.    5
Line no.    6
Line no.    7
\end{lstlisting}
```

→ Okay. And how can I get decreasing line numbers? Sorry, what? Decreasing line numbers as on page 37. 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 4.3.10 heavily. And do this step by step! For example, begin with '`framerule=10mm`'. If the rules are misaligned by the same (small) amount as before, the problem does not come from the rule width. So continue with the next parameter. Also, Adobe Acrobat sometimes has single-pixel rounding errors which can cause small misalignments at the corners when PDF files are displayed on screen; these are unfortunately normal.

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

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

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

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

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

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

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

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

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

Captions Now we come to `caption` and `label`. You might guess (correctly) that they can be used in the same manner as LaTeX's `\caption` and `\label` commands, although here it is also possible to have a caption regardless of whether or not the listing is in a float:[2]

```
\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.

---

[2] You should keep in mind that according to [Mi04] ... if the caption's text fits on one line, the text is centered; if the text does not fit on a single line, it will be typeset as a paragraph with a width equal to the line width.

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 8.

```
\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 have no width.

2.8 Emphasize identifiers

Recall the pretty-printing commands and environment. `\lstinline` prints code snippets, `\lstinputlisting` whole files, and `lstlisting` pieces of code which reside in the LaTeX file. And what are these different 'types' of source code good for? Well, it just happens that a sentence contains a code fragment. Whole files are typically included in or as an appendix. Nevertheless some books about programming also include such listings in normal text sections—to increase the number of pages. Nowadays source code should be shipped on disk or CD-ROM and only the main header or interface files should be typeset for reference. So, please, don't misuse the 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 who know the programming language well. The main matter is the presentation of interface, library or other functions or variables. If this is your concern, here come the right keys. Let's say, you want to emphasize the functions `square` and `root`, for example, by underlining them. Then you could do it like this:

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

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

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

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

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

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

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

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

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

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

2.9 Indexing

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

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

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

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

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

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

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

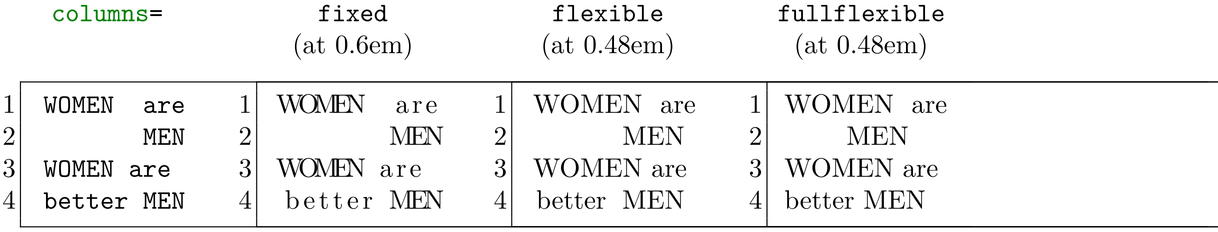
The `index` key has in fact two optional arguments. The first is the well-known ⟨*class number*⟩, the second is a comma separated list of other keyword classes whose identifiers are indexed. The indexed identifiers then change automatically with the defined keywords—not automagically, it's not an illusion.

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

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

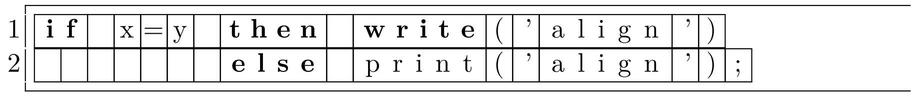
2.10 Fixed and flexible columns

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



→ 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



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 three formats is that the full flexible format cares about nothing else, while the normal flexible and space-flexible formats try to fix the column alignment if a character string needs less space than 'reserved'. The normal flexible format will insert make-up space to fix the alignment at spaces, before and after identifiers, and before and after sequences of other characters; the space-flexible format will only insert make-up space by stretching existing spaces. In the flexible example above, the two MENs are vertically aligned since some space has been inserted in the fourth line to fix the alignment. In the full flexible format, the two MENs are not aligned.

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

3 Advanced techniques

3.1 Style definitions

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

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

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

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

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

3.2 Language definitions

These are like style definitions except for an optional dialect name and an optional base language—and, of course, a different command name and specialized keys. In the simple case it's `\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 **b**ackslash or it is **d**oubled. 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—unless you specify something different. The *optional* `[`⟨*style*⟩`]` argument follows the *mandatory* `[`⟨*type*⟩`]` argument.

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

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

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

As you can see, you have the choice between specifying the style explicitly by LaTeX commands or implicitly by other style keys. But, you're right, some implicitly defined styles have no 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 to be applied cumulatively.

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

```
/* begin
    '␣string␣'
    typewriter */

 begin
 '␣string␣'
 /*typewriter*/
```

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

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

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

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

3.4 Closing and credits

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

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

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

Moreover we wish to thank

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

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.

Regarding the parameters, please keep in mind the following:

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

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

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

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

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

4.2 Typesetting listings

`\lstset{`⟨*key=value list*⟩`}`

sets the values of the specified keys, see also section 2.3. The parameters keep their values up to the end of the current group. In contrast, 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 unless requested differently in the optional argument, and do not support frames or background colors. You can write '`\lstinline!var i:integer;!`' and get '**var** i:**integer**;'.

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

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

Another limitation is that this feature can't be used in cells of a `tabular`-environment. See section 7.1 for a workaround.

See also section 4.6 for commands to create short analogs for the `\lstinline` command.

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

`\end{lstlisting}`

typesets the code in between as a displayed listing.

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

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

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

4.3 Options

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

4.3.1 Searching for files

`inputpath=`⟨*path*⟩ `{}`

defines the path, where the file given by ⟨*file name*⟩ resides.

`inputpath` overrules the `TEXINPUTS` environment variable, which means that a file residing on one of the paths given by `TEXINPUTS` isn't found anymore, if ⟨*path*⟩ isn't part of `TEXINPUTS`.

`inputpath` set as option of `\lstinputlisting` overrules the value set by `\lstset`.

4.3.2 Space and placement

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

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

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

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

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

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

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

define the space above and below displayed listings.

† `lineskip=`⟨*dimension*⟩ — `0pt`

specifies additional space between lines in listings.

† `boxpos=`⟨`b`|`c`|`t`⟩ — `c`

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

4.3.3 The printed range

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

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

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

`firstline=`⟨*number*⟩ — `1`

`lastline=`⟨*number*⟩ — `9999999`

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

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

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

In fact each part of the triplet ⟨*first*⟩`-`⟨*last*⟩ may be omitted. Omitting ⟨*first*⟩ or ⟨*last*⟩ means, that the range begins at line no. 1 and ends at ⟨*last*⟩ or begins at ⟨*first*⟩ ends at the last line of the file, a single number means, that the range begins and ends at the line given by the number, whereas omitting ⟨*first*⟩ or ⟨*last*⟩ means the range begins at line no. 1 and ends at the last line of the file.

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

can be used on individual listings only. Its use makes sense only if also `linerange` is used. The default (true) value means that the line numbering for *all* lineranges happens to be consecutively, e g. 1, 2, 3,. . . . If it is set to false, different ranges get their own numbering (see sec. 2.6).

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

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

`emptylines=`[`*`]⟨*number*⟩

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

`gobble=`⟨*number*⟩ `0`

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

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

4.3.4 Languages and styles

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

`style=`⟨*style name*⟩ `{}`

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

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

stores the key=value list.

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

activates a (dialect of a) programming language. The ‘empty’ default language detects no keywords, no comments, no strings, and so on; it may be useful for typesetting plain text. If ⟨*dialect*⟩ is not specified, the package chooses the default dialect, or the empty dialect if there is no default dialect.

Table 1 on page 14 lists all languages and dialects provided by `lstdrvrs.dtx`. The predefined default dialects are underlined.

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

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

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

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

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

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

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

prints or drops PODs in Perl.

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

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

optional `tagstyle=`⟨*style*⟩ `{}`

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

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

prints the first name in tags with keyword style.

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

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

4.3.5 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*⟩ `{}`

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

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

is used to print keywords. The optional ⟨*number*⟩ argument is the class number to which the style should be applied.

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

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

is equivalent to `keywordstyle=2`⟨*style*⟩.

classoffset=⟨*number*⟩ 0

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

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

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

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

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

optional directivestyle=⟨*style*⟩ `keywordstyle`

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

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

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

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

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

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

emphstyle=[⟨*number*⟩]{⟨*style*⟩}

respectively define, add or remove the ⟨*identifier list*⟩ from 'emphasize class ⟨*number*⟩', or 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)*⟩

define, add, or remove user supplied delimiters. (Note that this does not affect strings or comments.)

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

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

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

4.3.6 Getting all characters right

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

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

`inputencoding=`⟨*encoding*⟩ `{}`

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

`upquote=`⟨`true`|`false`⟩ `false`

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

`tabsize=`⟨*number*⟩ `8`

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

`showtabs=`⟨`true`|`false`⟩ `false`

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

`tab=`⟨*tokens*⟩

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

`showspaces=`⟨`true`|`false`⟩ `false`

lets all blank spaces appear ␣ or as blank spaces.

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

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

`formfeed=`⟨*tokens*⟩ `\bigbreak`

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

4.3.7 Line numbers

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

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

`stepnumber=`⟨*number*⟩ `1`

All lines with “line number ≡ 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 (logical) number.

`name=`⟨*name*⟩

names a listing. Displayed environment-listings with the same name share a line counter if `firstnumber=auto` is in effect.

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

prints the lines’ numbers.

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

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

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

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






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

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

4.3.8 Captions

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

`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 4.3.7.

`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 `\lstlistingnamestyle` `{}`

customizes the style of the caption label for program listings in a simple way, something like `\small`, `\bfseries` or a combination of several commands. If there is a need for a complete customization of the label (justification, fonts, margins, ...), one should use the `caption` package by A. Sommerfeldt [?].

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

prints the running number of the caption.

`numberbychapter=`⟨`true`|`false`⟩ `true`

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

`\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 respectively above or below each caption.

4.3.9 Margins and line shape

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

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

`xleftmargin=`⟨*dimension*⟩ `0pt`

`xrightmargin=`⟨*dimension*⟩ `0pt`

The dimensions are used as extra margins on the left and right. Line numbers and frames are both moved accordingly.

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

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

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

activates or deactivates automatic line breaking of long lines.

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

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

`prebreak=`⟨*tokens*⟩ `{}`

`postbreak=`⟨*tokens*⟩ `{}`

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

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

`breakindent=`⟨*dimension*⟩ `20pt`

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

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

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

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

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

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

```
               { Now auto ⤸
  indention is off. }
```

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

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

4.3.10 Frames

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

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

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

`frame=`⟨*subset of* `trblTRBL`⟩ `{}`

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

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

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

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

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

`framesep=`⟨*dimension*⟩ `3pt`

`rulesep=`⟨*dimension*⟩ `2pt`

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

`framerule=`⟨*dimension*⟩ `0.4pt`

controls the width of the rules.

`framexleftmargin=`⟨*dimension*⟩ `0pt`

`framexrightmargin=`⟨*dimension*⟩ `0pt`

`framextopmargin=`⟨*dimension*⟩ `0pt`

`framexbottommargin=`⟨*dimension*⟩ `0pt`

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

`backgroundcolor=`⟨*color command*⟩

`rulecolor=`⟨*color command*⟩

`fillcolor=`⟨*color command*⟩

`rulesepcolor=`⟨*color command*⟩

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

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

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

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

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

Note here the use of `framexleftmargin` to include the line numbers inside the frame.

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

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

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

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

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

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

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

To summarize, the example above used

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

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

4.3.11 Indexing

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

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

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

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

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

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

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

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

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

Section 2.9 describes this feature in detail.

4.3.12 Column alignment

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

selects the column alignment. The ⟨*alignment*⟩ can be `fixed`, `flexible`, `spaceflexible`, or `fullflexible`; see section 2.10 for details.

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

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

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

† `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 4.3.5 make heavy font size changes, see the example below.

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

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

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

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

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

4.3.13 Escaping to LaTeX

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

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

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

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

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

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

`mathescape=`⟨`true`|`false`⟩ `false`

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

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

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

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

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

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

`escapebegin=`⟨*tokens*⟩ `{}`

`escapeend=`⟨*tokens*⟩ `{}`

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

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

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

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

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

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

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

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

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

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

Expand that list yourself and mail me about new items.

4.4 Interface to fancyvrb

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

`fancyvrb`=⟨`true`|`false`⟩

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

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

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

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

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

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

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

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

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

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

\par\vspace{72.27pt}

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

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

4.5 Environments

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

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

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

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

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

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

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

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

4.6 Short Inline Listing Commands

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

`\lstMakeShortInline`[`[`⟨*options*⟩`]`]⟨*character*⟩

defines ⟨*character*⟩ to be an equivalent of `\lstinline`[`[`⟨*options*⟩`]`]⟨*character*⟩, allowing for a convenient syntax when using lots of inline listings.

`\lstDeleteShortInline`⟨*character*⟩

removes a definition of ⟨*character*⟩ created by `\lstMakeShortInline`, and returns ⟨*character*⟩ to its previous meaning.

4.7 Language definitions

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

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

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

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

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

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

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

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

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

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

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

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

†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 a time. (3) If used 'standalone' outside a language definition, the key might work only after selecting a nonempty language (and switching back to the empty language if necessary). (4) The key does not respect the value of `classoffset` and has no optional class ⟨*number*⟩ argument.

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

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

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

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

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

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

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

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

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

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

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

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

Ditto for control sequences in TeX and LaTeX.

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

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

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.

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

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

`sensitive=`⟨`true`|`false`⟩

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

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

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

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

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

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

`otherkeywords={`⟨*keywords*⟩`}`

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

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

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

Strings

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

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

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

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

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

Comments

`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 example, but the examples apply to `comment` and `deletecomment` as well.

`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]`⟨*delimiter*⟩

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

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

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

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

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

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

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

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

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

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

activates or deactivates PODs—Perl specific.

4.8 Installation

Software installation

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

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

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

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

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

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

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

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

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

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

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

`lstmisc0.sty`	for	local add-ons (see the developer's guide),
`lstlang0.sty`	for	local language definitions (see 4.7), and
`lstlocal.cfg`	as	local configuration file

and put them in the same directory as the other listings files. These three files are not touched by a new installation unless 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`, `\lstlistingnamestyle`, `defaultdialect`, `\lstalias`, or `\lstalias` as described in earlier sections.

5 Experimental features

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

5.1 Listings inside arguments

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

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

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

yields[3] if the current language is Pascal. Note that this example shows another experimental feature: use of argument braces as delimiters. This is described in section 4.2.

And now an environment example:

```
 !”#$%&’()*+,−./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]ˆ_
‘abcdefghijklmno
pqrstuvwxyz{|}˜
```

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

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

5.2 † Export of identifiers

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

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

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

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

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

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

---

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

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

defines the style in which procedure and function names appear.

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

If activated, procedure and function names are also indexed.

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

5.3 † Hyperlink references

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

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

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

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

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

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

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

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

5.4 Literate programming

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

```
var  i:integer;

if  (i≤0)  i ← 1;
if  (i≥0)  i ← 0;
if  (i≠0)  i ← 0;
```

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

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

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

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

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

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

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

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

In the example above, I've used

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

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

5.5 LGrind definitions

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

optional `lgrindef`=⟨*language*⟩

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

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

data,optional `\lstlgrindeffile` `lgrindef.`

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

5.6 † Automatic formatting

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

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

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

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

Now we can give an example.

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

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

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

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

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

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

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

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

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

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

5.7 Arbitrary linerange markers

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

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

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

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

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

`rangebeginprefix`=⟨*prefix*⟩

`rangebeginsuffix`=⟨*suffix*⟩

`rangeendprefix`=⟨*prefix*⟩

`rangeendsuffix`=⟨*suffix*⟩

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

`rangeprefix`=⟨*prefix*⟩

`rangesuffix`=⟨*suffix*⟩

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

`includerangemarker`=⟨`true`|`false`⟩ `true`

shows or hides the markers in the output.

Remark: If `firstnumber` is set, it refers to the line which contains the marker. So if one wants to start a range with the number 1, one has to set `includerangemarker=false, firstnumber=0`.

```
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}
```

5.8 Multicolumn Listings

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

```
if (i < 0)          end if
  i = 0             if (j < 0)
  j = 1               j = 0
                    end if
```

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

The multicolumn option is known to fail with some keys.

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

Tips and tricks

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

6 Troubleshooting

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

1. Start from the minimal file in section 1.1.
2. Add the LaTeX code which causes the problem, but keep it short. In particular, keep the number of additional packages small.

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

7 Bugs and workarounds

7.1 Listings inside arguments

At the moment it isn't possible to use `\lstinline{...}` in a cell of a table (see section 18.4.1 on page 200 for more information), but it is possible to define a wrapper macro which can be used instead of `\lstinline{...}`:

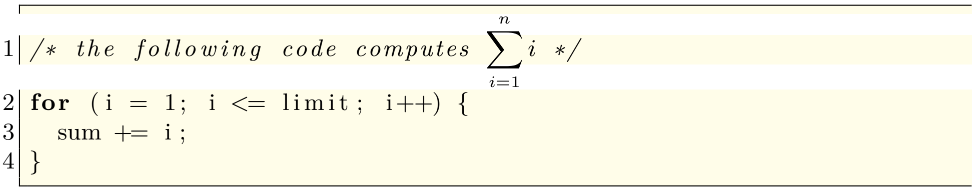
```
\newcommand\foo{\lstinline{t}}
\newcommand\foobar[2][]{\lstinline[#1]{#2}}

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

t	a variable
int u;	a declaration

7.2 Listings with a background colour and LaTeX escaped formulas

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

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



/ the following code computes* $\displaystyle\sum_{i=1}^{n} i$ */*

```
for (i = 1; i <= limit; i++) {
  sum += i;
}
```

At the moment there is only one workaround:

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

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

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

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

8 How tos

How to reference line numbers

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

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

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

Line 3 shows a comment.

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

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

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

How to gobble characters

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

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

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

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

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

Note that empty lines and the beginning and the end of the environment need not respect the indention. However, 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 would never have had.

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

The following example is originally from a header file I got from Herbert. For the presentation here I use the `lstlisting` environment and an excerpt from the header file. The `\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, because the graphic wasn't available when the manual was typeset.

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 4.3.13. The keys `escapechar`, `escapeinside`, and `texcl` allow partial usage of LaTeX code.

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

```
%    \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 work with plain text

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

How to get the developer's guide

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

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

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

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

```
%    make all
```

or

```
%    make listings-devel.pdf
```

or

```
%    make pdf-devel
```

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

Developer's guide

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

9 Basic concepts

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

9.1 Package loading

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

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`

writefile requires 1 `\toks`, 1 `\write`

`\lst@BeginWriteFile`, `\lst@BeginAlsoWriteFile`, `\lst@EndWriteFile`

style

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

empty language, language, alsolanguage, 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`, `consecutivenumbers`, `nolol`, `captionpos`, `abovecaptionskip`, `belowcaptionskip`, `label`, `title`, `caption`, `\lstlistingname`, `\lstlistingnamestyle`, `boxpos`, `float`, `floatplacement`, `aboveskip`, `belowskip`, `everydisplay`, `showlines`, `emptylines`, `gobble`, `name`, `\lstname`, `\lstlistlistingname`, `\lstlistoflistings`, `\lstnewenvironment`, `\lstinline`, `\lstinputlisting`, `lstlisting`, `\lstloadaspects`, `\lstset`, `\thelstlisting`, `\lstaspectfiles`, `inputencoding`, `inputpath`, `delim`, `moredelim`, `deletedelim`, `upquote`, `numberbychapter`, `\lstMakeShortInline`, `\lstDeleteShortInline`, `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. ⟨*aspect name*⟩ 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. ⟨*definition*⟩ is the replacement text of a macro with one parameter. The argument is either the value from 'key=value' or ⟨*default value*⟩ if no '=value' is given. The helper macros `\lstKV@...` below might simplify ⟨*definition*⟩.

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

`\lst@AddToHook{`⟨*name of hook*⟩`}{`⟨*TeX material*⟩`}`

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

`\lst@AddToHookExe{`⟨*name of hook*⟩`}{`⟨*TeX material*⟩`}`

also executes ⟨*TeX material*⟩ for initialization. You might use local variables—local in the sense of TeX and/or usual programming languages—but when the code is executed for initialization all assignments are global: we set `\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 ⟨*definition*⟩ argument of the `\lst@Key` command to evaluate the argument. The additional prefix `KV` refers to the keyval package.

`\lstKV@SetIf{`⟨*value*⟩`}`⟨*if macro*⟩

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

`\lstKV@SwitchCases{`⟨*value*⟩`}`
`{`⟨*string 1*⟩`&`⟨*execute 1*⟩`\\`
⟨*string 2*⟩`&`⟨*execute 2*⟩`\\`
⋮
⟨*string n*⟩`&`⟨*execute n*⟩`}{`⟨*else*⟩`}`

Either execute ⟨*else*⟩ or the ⟨*value*⟩ matching part.

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

`\lstKV@TwoArg{`⟨*value*⟩`}{`⟨*subdefinition*⟩`}`

`\lstKV@ThreeArg{`⟨*value*⟩`}{`⟨*subdefinition*⟩`}`

`\lstKV@FourArg{`⟨*value*⟩`}{`⟨*subdefinition*⟩`}`

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

`\lstKV@OptArg[`⟨*default arg.*⟩`]{`⟨*value*⟩`}{`⟨*subdefinition*⟩`}`

`[`⟨*default arg.*⟩`]` is *not* optional. ⟨*subdefinition*⟩ 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. ⟨*subdefinition*⟩ accesses these arguments via `##1` and `##2`.

⟨*value*⟩ is usually the argument `#1` passed by the keyval package. If ⟨*value*⟩ has no optional argument, ⟨*default arg.*⟩ is inserted to provide the arguments to ⟨*subdefinition*⟩.

`\lstKV@XOptArg[`⟨*default arg.*⟩`]{`⟨*value*⟩`}`⟨*submacro*⟩

Same as `\lstKV@OptArg` but the third argument ⟨*submacro*⟩ 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@ifsensitive}
%    \lst@AddToHookExe{SetLanguage}{\let\lst@ifsensitive\iftrue}
```

The last line is equivalent to

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

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

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

9.3 Internal modes

You probably know TEX's conditional commands `\ifhmode`, `\ifvmode`, `\ifmmode`, and `\ifinner`. They tell you whether TEX is in (restricted) horizontal or (internal) vertical or in (nondisplay) mathematical mode. For example, true `\ifhmode` and true `\ifinner` indicate restricted horizontal mode, which means that you are in a `\hbox`. The typical user doesn't care about such modes; TEX/LATEX 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 LATEX works since 'no mode' implies that TEX's characters and catcodes are present, whereas 'processing mode' means that listings' characters and catcodes are active.

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

`\lst@NewMode`⟨*mode* (*control sequence*)⟩

defines a new static mode, which is a nonnegative integer assigned to ⟨*mode*⟩. ⟨*mode*⟩ should have the prefix `lst@` and suffix `mode`.

`\lst@UseDynamicMode{`⟨*token*(*s*)⟩`}`

inserts a dynamic mode number as argument to the token(s).

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

Table 3: Internal modes

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

changed `\lst@EnterMode`⟨*mode*⟩`{`⟨*start tokens*⟩`}`

opens a group level, enters the mode, and executes ⟨*start tokens*⟩.

Use `\lst@modetrue` in ⟨*start tokens*⟩ to prohibit future mode changes—except leaving the mode, of course. You must test yourself whether you're allowed to enter, see below.

`\lst@LeaveMode`

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

`\lst@InterruptModes`

`\lst@ReenterModes`

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

Remember that `\lst@nomode` implies that TEX's character table is active.

Some variables show the internal state of processing. You are allowed to read them, but *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=`⟨*mode name*⟩ 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 ⟨*start tokens*⟩.

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 TEX's character table. First we define a standard character table which contains

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

This character table is altered depending on the current programming language. We may define string and comment delimiters or other special characters. Table 2 on page 48 shows the standard character table. It can be modified with the keys `alsoletter`, `alsodigit`, and `alsoother`.

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

```
%    \def A{\lst@ProcessLetter A}
```

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

`\lst@ProcessLetter` ⟨*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 TEX's ord-operator ` on such an argument, e.g. don't write `` \ifnum`#1=65 `` to test against '`A`'.

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

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

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

9.6 On the output

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

data `\lst@lastother`

equals ⟨*spec. token*⟩ version of the last processed nonidentifier-character. Since programming languages redefine the standard character table, we use the original ⟨*spec. token*⟩. 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 ⟨*spec. token*⟩ expands either to a printable or to a token character.

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

`\lst@outputspace`

Use this predefined ⟨*spec. token*⟩ (obviously for character code 32) to test against `\lst@lastother`.

`\lstum@backslash`

Use this predefined ⟨*spec. token*⟩ (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{`⟨*character code*⟩`}`⟨*macro*⟩

Stores the ⟨*spec. token*⟩ corresponding to ⟨*character code*⟩ in ⟨*macro*⟩. 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 ⟨*character code*⟩ 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`

`{`⟨*prefix*⟩`}`	`k`
`{`⟨*name*⟩`}`	`{keywords}`
`{`⟨*style name*⟩`}`	`{keywordstyle}`
`{`⟨*style init*⟩`}`	`\bfseries`
`{`⟨*default style name*⟩`}`	`{keywordstyle}`
`{`⟨*working procedure*⟩`}`	`{}`
⟨`l`|`o`⟩	`l`
⟨`d`|`o`⟩	`d`

installs either a keyword or 'working' class of identifiers according to whether ⟨*working procedure*⟩ is empty.

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

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

`l` 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 ⟨`d`|`o`⟩.

`\lst@InstallKeywords`

`{`⟨*prefix*⟩`}`	`cs`
`{`⟨*name*⟩`}`	`{texcs}`
`{`⟨*style name*⟩`}`	`{texcsstyle}`
`{`⟨*style init*⟩`}`	`\relax`
`{`⟨*default style name*⟩`}`	`{keywordstyle}`
`{`⟨*working procedure*⟩`}`	see below
⟨`l`|`o`⟩	`l`
⟨`d`|`o`⟩	`d`

Same parameters, same functionality with one execption. The macro installs exactly one keyword class and not a whole family. Therefore the argument to ⟨*working procedure*⟩ 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 `l`, but there is no string which starts with the given delimiter and ends at end of line. So we'll add it now!

First of all we extend the list of string types by

```
%    \lst@AddTo\lst@stringtypes{,l}
```

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@l#1#2\@empty#3#4#5{%
%        \lst@CArg #2\relax\lst@DefDelimB{}{}{}#3{#1}{#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!

```
//␣This␣is␣a␣string.
This isn't a string.
```

```
\lstset{string=[l]//}
\begin{lstlisting}
// This is 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 `{`⟨*style*⟩`}`⟨*start tokens*⟩. This with a preceding `\def\lst@currstyle` is used as argument to `\lst@EnterMode`. The delimiter class also provides it. In the example we 'extended' `#5` by `\lst@Lmodetrue` (line mode true). The mode automatically ends at end of line, so we didn't need the end-delimiter argument.

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

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

`\lst@DefDelimB`

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

defines ⟨*1st*⟩⟨*2nd*⟩⟨*rest*⟩ as starting-delimiter. ⟨*execute*⟩ is executed when the package comes to ⟨*1st*⟩. ⟨*delim exe modetrue*⟩ and ⟨*delim exe modefalse*⟩ are executed only if the whole delimiter ⟨*1st*⟩⟨*2nd*⟩⟨*rest*⟩ is found. Exactly one of them is called depending on `\lst@ifmode`.

By default the package enters the mode if the delimiter is found *and* `\lst@ifmode` is false. Internally we make an appropriate definition of `\lst@bnext`, which can be gobbled by placing `\@gobblethree` at the very end of ⟨*delim exe modefalse*⟩. One can provide an own definition (and gobble the default).

⟨*save 1st*⟩ must be an undefined macro and is used internally to store the previous meaning of ⟨*1st*⟩. The arguments ⟨*2nd*⟩ and/or ⟨*rest*⟩ are empty if the delimiter has strictly less than three characters. All characters of ⟨*1st*⟩⟨*2nd*⟩⟨*rest*⟩ must already be active (if not empty). That's not a problem since the macro `\lst@CArgX` does this job.

`\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 4.5. New commands like `\lstinline` or `\lstinputlisting` are more difficult. Roughly speaking you must follow these steps.

1. Open a group to make all changes local.
2. ⟨*Do whatever you want.*⟩
3. Call `\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`⟨*macro*⟩`{`⟨*TeX material*⟩`}`

adds ⟨*TeX material*⟩ globally to the contents of ⟨*macro*⟩.

`\lst@Extend`⟨*macro*⟩`{`⟨*TeX material*⟩`}`

calls `\lst@AddTo` after the first token of ⟨*TeX material*⟩ is `\expandedafter`. For example, `\lst@Extend \a \b` merges the contents of the two macros and stores it globally in `\a`.

`\lst@lAddTo`⟨*macro*⟩`{`⟨*TeX material*⟩`}`

`\lst@lExtend`⟨*macro*⟩`{`⟨*TeX material*⟩`}`

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

`\lst@DeleteKeysIn`⟨*macro*⟩⟨*macro* (*keys to remove*)⟩

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

`\lst@ReplaceIn`⟨*macro*⟩⟨*macro* (*containing replacement list*)⟩

`\lst@ReplaceInArg`⟨*macro*⟩`{`⟨*replacement list*⟩`}`

The replacement list has the form $a_1b_1\ldots a_nb_n$, where each a_i and b_i is a character sequence (enclosed in braces if necessary) and may contain macros, but the first token of b_i must not be equivalent to `\@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{`⟨*character sequence*⟩`}`⟨*macro*⟩`{`⟨*then*⟩`}{`⟨*else*⟩`}`

⟨*then*⟩ is executed if ⟨*character sequence*⟩ is a substring of the contents of ⟨*macro*⟩. Otherwise ⟨*else*⟩ is called.

`\lst@IfOneOf`⟨*character sequence*⟩`\relax`⟨*macro*⟩`{`⟨*then*⟩`}{`⟨*else*⟩`}`

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

`\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 LaTeX'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 (`{}\`); 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 TEX since such catcodes can be changed easily. It is explicitly allowed that the charcaters have been read, e.g. in `\def\test{\lst@DefActive\temp{ABC}}`!

Note that this macro changes `\lccode`s 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=`⟨*TEX code*⟩

`MoreSelectCharTable=`⟨*TEX code*⟩

uses ⟨*TEX code*⟩ (additionally) to select the character table. The code is executed after the standard character table is selected, but possibly before other aspects make more changes. Since previous meanings are always saved and executed inside the new definition, this should be harmless.

Here come two rather useless examples. Each point (full stop) will cause a message '`.`' on the terminal and in the `.log` file if language `useless` is active:

```
%   \lstdefinelanguage{useless}
%       {SelectCharTable=\lst@DefSaveDef{46}% save chr(46) ...
%            \lsts@point             % ... in \lsts@point and ...
%            {\message{.}\lsts@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}}
%   \lstdefinelanguage[2]{useless}
%       {SelectCharTable=\lst@DefSaveDef{46}\lsts@point
%            {\global\advance\lst@points\@ne \lsts@point}
%       }
```

`% \global` indicates that the allocated counter is used globally. We zero the counter at the beginning of each listing, display a message about the current value at the end of a listing, and each processed point advances the counter by one.

`\lst@CArg`⟨*active characters*⟩`\relax`⟨*macro*⟩

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

`\lst@CArgX`⟨*characters*⟩`\relax`⟨*macro*⟩

> makes ⟨*characters*⟩ active before calling `\lst@CArg`.

`\lst@CDef{`⟨*1st*⟩⟨*2nd*⟩`{`⟨*rest*⟩`}}`⟨*save 1st*⟩`{`⟨*execute*⟩`}{`⟨*pre*⟩`}{`⟨*post*⟩`}`

> should be used in connection with `\lst@CArg` or `\lst@CArgX`, i.e. as ⟨*macro*⟩ there. ⟨*1st*⟩, ⟨*2nd*⟩, and ⟨*rest*⟩ must be active characters and ⟨*save 1st*⟩ must be an undefined control sequence.
>
> Whenever the package reaches the character ⟨*1st*⟩ (in a listing), ⟨*execute*⟩ is executed. If the package detects the whole string ⟨*1st*⟩⟨*2nd*⟩⟨*rest*⟩, we additionally execute ⟨*pre*⟩, then the string, and finally ⟨*post*⟩.

`\lst@CDefX`⟨*1st*⟩⟨*2nd*⟩`{`⟨*rest*⟩`}`⟨*save 1st*⟩`{`⟨*execute*⟩`}{`⟨*pre*⟩`}{`⟨*post*⟩`}`

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

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 `\lstMakeShortInline@` does the main work for `\lstMakeShortInline`.

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

```
⟨*kernel | misc⟩
%% Please read the software license in listings.dtx or listings.pdf.
%%
%% (w)(c) 1996--2004 Carsten Heinz and/or any other author listed
%% elsewhere in this file.
%% (c) 2006 Brooks Moses
%% (c) 2013- Jobst Hoffmann
%%
%% Send comments and ideas on the package, error reports and additional
%% programming languages to Jobst Hoffmann at <j.hoffmann@fh-aachen.de>.
%%
⟨/kernel | misc⟩
```

Identification All files will have same date and version.

```
⟨*kernel | misc | doc⟩
\def\filedate{2023/02/27}
\def\fileversion{1.9}
⟨/kernel | misc | doc⟩
```

What we need and who we are.

```
⟨*kernel⟩
\NeedsTeXFormat{LaTeX2e}
\AtEndOfPackage{\ProvidesPackage{listings}
             [\filedate\space\fileversion\space(Carsten Heinz)]}
```

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

```
\def\lst@CheckVersion#1{\edef\reserved@a{#1}%
    \ifx\lst@version\reserved@a \expandafter\@gobble
                          \else \expandafter\@firstofone \fi}
```

```
\let\lst@version\fileversion
⟨/kernel⟩
```

For example by the miscellaneous file

```
⟨*misc⟩
\ProvidesFile{lstmisc.sty}
             [\filedate\space\fileversion\space(Carsten Heinz)]
\lst@CheckVersion\fileversion
    {\typeout{^^J%
     ***^^J%
     *** This file requires ‘listings.sty’ version \fileversion.^^J%
     *** You have a serious problem, so I’m exiting ...^^J%
     ***^^J}%
     \batchmode \@@end}
⟨/misc⟩
```

or by the dummy patch.

```
⟨*patch⟩
\ProvidesFile{lstpatch.sty}
             [\filedate\space\fileversion\space(Carsten Heinz)]
\lst@CheckVersion\lst@version{}
⟨/patch⟩
```

```
⟨*doc⟩
\ProvidesPackage{lstdoc}
             [\filedate\space\fileversion\space(Carsten Heinz)]
⟨/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.

```
⟨*kernel⟩
\def\lst@InputCatcodes{%
    \makeatletter \catcode`\"12%
    \catcode`\^^@\active
    \catcode`\^^I9%
    \catcode`\^^L9%
    \catcode`\^^M9%
    \catcode`\%14%
    \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.

```
\def\lst@RestoreCatcodes#1{%
    \ifx\relax#1\else
        \noexpand\catcode`\noexpand#1\the\catcode`#1\relax
        \expandafter\lst@RestoreCatcodes
    \fi}
\edef\lst@RestoreCatcodes{%
    \noexpand\lccode`\noexpand\/`\noexpand\/%
    \lst@RestoreCatcodes\"\^^I\^^M\~\^^@\relax
    \catcode12\active}
```

Now we are ready for

```
\lst@InputCatcodes
\AtEndOfPackage{\lst@RestoreCatcodes}
⟨/kernel⟩
```

Statistics

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

```
⟨*info⟩
\def\lst@GetAllocs{%
    \edef\lst@allocs{%
        0\noexpand\count\the\count10,1\noexpand\dimen\the\count11,%
        2\noexpand\skip\the\count12,3\noexpand\muskip\the\count13,%
        4\noexpand\box\the\count14,5\noexpand\toks\the\count15,%
        6\noexpand\read\the\count16,7\noexpand\write\the\count17}}
\def\lst@ReportAllocs{%
    \message{^^JAllocs:}\def\lst@temp{none}%
    \expandafter\lst@ReportAllocs@\lst@allocs,\z@\relax\z@,}
\def\lst@ReportAllocs@#1#2#3,{%
    \ifx#2\relax \message{\lst@temp^^J}\else
        \@tempcnta\count1#1\relax \advance\@tempcnta -#3\relax
        \ifnum\@tempcnta=\z@\else
            \let\lst@temp\@empty
            \message{\the\@tempcnta \string#2,}%
        \fi
        \expandafter\lst@ReportAllocs@
    \fi}
\lst@GetAllocs
```

```
⟨/info⟩
```

Miscellaneous

`\@lst` Just a definition to save memory space.

```
⟨*kernel⟩
\def\@lst{lst}
⟨/kernel⟩
```

13 General problems

All definitions in this section belong to the kernel.

```
⟨*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.

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

`\lst@IfOneOf` Ditto.

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

```
\def\lst@DeleteKeysIn#1#2{%
    \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.

```
\def\lst@DeleteKeysIn@#1#2,{%
    \ifx\relax#2\@empty
```

```
        \expandafter\@firstoftwo\expandafter\lst@RemoveCommas
    \else
        \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.

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

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

```
%            \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`.

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

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

```
%\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`
`\lst@ReplaceInArg`

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

```
\def\lst@ReplaceIn#1#2{%
    \expandafter\lst@ReplaceIn@\expandafter#1#2\@empty\@empty}
\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).

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

13.2 Flow of control

`\@gobblethree` is defined if and only if undefined.

```
\providecommand*\@gobblethree[3]{}
```

`\lst@GobbleNil`

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

`\lst@Swap` is just this:

```
\def\lst@Swap#1#2{#2#1}
```

`\lst@if` `\lst@true` `\lst@false` A general `\if` for temporary use.

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

`\lst@IfNextCharsArg` is quite easy: We define a macro and call `\lst@IfNextChars`.

```
\def\lst@IfNextCharsArg#1{%
    \def\lst@tofind{#1}\lst@IfNextChars\lst@tofind}
```

`\lst@IfNextChars` We save the arguments and start a loop.

```
\def\lst@IfNextChars#1#2#3{%
    \let\lst@tofind#1\def\@tempa{#2}\def\@tempb{#3}%
    \let\lst@eaten\@empty \lst@IfNextChars@}
```

Expand the characters we are looking for.

```
\def\lst@IfNextChars@{\expandafter\lst@IfNextChars@@\lst@tofind\relax}
```

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

```
\def\lst@IfNextChars@@#1#2\relax#3{%
    \def\lst@tofind{#2}\lst@lAddTo\lst@eaten{#3}%
    \ifx#1#3%
```

If characters are the same, we either call `\@tempa` or continue the test.

```
        \ifx\lst@tofind\@empty
            \let\lst@next\@tempa
        \else
            \let\lst@next\lst@IfNextChars@
        \fi
        \expandafter\lst@next
    \else
```

If the characters are different, we call `\@tempb`.

```
        \expandafter\@tempb
    \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`.

```
\def\lst@IfNextCharActive#1#2#3{%
```

```
    \begingroup \lccode`\~=`#3\lowercase{\endgroup
    \ifx~}#3%
        \def\lst@next{#1}%
    \else
        \def\lst@next{#2}%
    \fi \lst@next #3}
```

`\lst@for` A for-loop with expansion of the loop-variable. This was improved due to a suggestion by Hendri Adriaens.

```
\def\lst@for#1\do#2{%
  \def\lst@forbody##1{#2}%
  \def\@tempa{#1}%
  \ifx\@tempa\@empty\else\expandafter\lst@f@r#1,\@nil,\fi
}
\def\lst@f@r#1,{%
  \def\@tempa{#1}%
  \ifx\@tempa\@nnil\else\lst@forbody{#1}\expandafter\lst@f@r\fi
}
```

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 `#1` is the lower case version of `~`. Fortunately the `\lowercase` doesn't change the catcode, so we have an active version of `#1`. Note that `~` is usually active.

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

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

We get the argument, empty `\lst@arg` and begin a loop.

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

There are nine `\relax`es 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`.

```
\begingroup
\catcode`\^^@=\active \catcode`\^^A=\active \catcode`\^^B=\active
\catcode`\^^C=\active \catcode`\^^D=\active \catcode`\^^E=\active
\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.

```
\gdef\lst@MakeActive@#1#2#3#4#5#6#7#8#9{\let\lst@next\relax
    \ifx#1\relax
    \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.

```
    \ifx#2\relax
        \lowercase{\lst@lAddTo\lst@temp{^^@}}%
    \else \lccode‘\^^A=‘#2%
    \ifx#3\relax
        \lowercase{\lst@lAddTo\lst@temp{^^@^^A}}%
    \else \lccode‘\^^B=‘#3%
    \ifx#4\relax
        \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B}}%
    \else \lccode‘\^^C=‘#4%
    \ifx#5\relax
        \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C}}%
    \else \lccode‘\^^D=‘#5%
    \ifx#6\relax
        \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C^^D}}%
    \else \lccode‘\^^E=‘#6%
    \ifx#7\relax
        \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C^^D^^E}}%
    \else \lccode‘\^^F=‘#7%
    \ifx#8\relax
        \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C^^D^^E^^F}}%
    \else \lccode‘\^^G=‘#8%
    \ifx#9\relax
        \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`.

```
    \else \lccode‘\^^H=‘#9%
        \lowercase{\lst@lAddTo\lst@temp{^^@^^A^^B^^C^^D^^E^^F^^G^^H}}%
        \let\lst@next\lst@MakeActive@
    \fi \fi \fi \fi \fi \fi \fi \fi \fi
    \lst@next}
\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`.

```
\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.

```
\def\lst@DefOther#1#2{%
    \begingroup \def#1{#2}\escapechar\m@ne \expandafter\endgroup
    \expandafter\lst@DefOther@\meaning#1\relax#1}
\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` If `mathescape` is not on, we call (near the end of this definition) a submacro similar to `\zap@space` to replace single spaces by active spaces. Otherwise we check whether the code contains a pair `$...$` and call the appropriate macro.

```
\def\lst@InsideConvert#1{%
   \lst@ifmathescape
      \lst@InsideConvert@e#1$\@nil
      \lst@if
         \lst@InsideConvert@ey#1\@nil
      \else
         \lst@InsideConvert@#1 \@empty
         \expandafter\@gobbletwo
      \fi
      \expandafter\lst@next
   \else
      \lst@InsideConvert@#1 \@empty
   \fi}
\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.

```
\gdef\lst@InsideConvert@#1 #2{%
    \lst@MakeActive{#1}%
    \ifx\@empty#2%
        \lst@lExtend\lst@arg{\lst@temp}%
    \else
        \lst@lExtend\lst@arg{\lst@temp~}%
        \expandafter\lst@InsideConvert@
    \fi #2}
```

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

```
}\endgroup
```

The next definition has been used above to check for `$...$` and the following one keeps the math contents from being converted. This feature was requested by Dr. Jobst Hoffmann.

```
\def\lst@InsideConvert@e#1$#2\@nil{%
   \ifx\@empty#2\@empty \lst@false \else \lst@true \fi}
\def\lst@InsideConvert@ey#1$#2$#3\@nil{%
   \lst@InsideConvert@#1 \@empty
   \lst@lAddTo\lst@arg{%
      \lst@ifdropinput\else
         \lst@TrackNewLines\lst@OutputLostSpace \lst@XPrintToken
         \setbox\@tempboxa=\hbox\bgroup$\lst@escapebegin
         #2%
         \lst@escapeend$\egroup \lst@CalcLostSpaceAndOutput
         \lst@whitespacefalse
      \fi}%
   \def\lst@next{\lst@InsideConvert{#3}}%
}
```

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

```
\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.

```
\def\lst@XConvertArg#1{%
    {\lst@false \let\lst@arg\@empty
     \lst@XConvert#1\@nil
     \global\let\@gtempa\lst@arg}%
    \lst@lExtend\lst@arg{\expandafter{\@gtempa}}%
    \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.

```
\def\lst@XConvert@#1{%
    \ifx\@nil#1\else
        \begingroup\lccode`\~=`#1\lowercase{\endgroup
        \lst@lAddTo\lst@arg~}%
        \expandafter\lst@XConvertNext
    \fi}
\def\lst@XConvertNext{%
    \lst@if \expandafter\lst@XConvertX
```

```
      \else \expandafter\lst@XConvert \fi}
```

Now we make only the first character active.

```
\def\lst@XConvertX#1{%
    \ifx\@nil#1\else
        \lst@XConvertX@#1\relax
        \expandafter\lst@XConvert
    \fi}
\def\lst@XConvertX@#1#2\relax{%
    \begingroup\lccode`\~=`#1\lowercase{\endgroup
    \lst@XCConvertX@@~}{#2}}
\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), ...

```
\def\lst@Require#1#2#3#4#5{%
    \begingroup
    \aftergroup\lst@true
    \ifx\@empty#3\@empty\else
        \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.

```
        \edef\lst@temp{\expandafter\zap@space#3 \@empty}%
        \lst@for\lst@temp\do{%
          \ifx\@empty##1\@empty\else \lstKV@OptArg[]{##1}{%
            #4[####1]{####2}%
            \@ifundefined{\@lst\lst@prefix @\lst@malias $\lst@oalias}%
            {\edef\lst@require{\lst@require,\lst@malias $\lst@oalias}}%
            {}}%
          \fi}%
```

Init things and input files if and as long as it is necessary.

```
        \global\let\lst@loadaspects\@empty
        \lst@InputCatcodes
        \ifx\lst@require\@empty\else
            \lst@for{#5}\do{%
                \ifx\lst@require\@empty\else
                    \InputIfFileExists{##1}{}{}%
                \fi}%
        \fi
```

Issue error and call `\lst@false` (after closing the local group) if some items weren't loadable.

```
        \ifx\lst@require\@empty\else
            \PackageError{Listings}{Couldn't load requested #1}%
            {The following #1s weren't loadable:^^J\@spaces
             \lst@require^^JThis may cause errors in the sequel.}%
            \aftergroup\lst@false
        \fi
```

Request aspects.

```
        \ifx\lst@loadaspects\@empty\else
            \lst@RequireAspects\lst@loadaspects
        \fi
    \fi
    \endgroup}
```

`\lst@IfRequired` uses `\lst@IfOneOf` and adds some code to ⟨*then*⟩ part: delete the now loaded item from the list and define `\lst`⟨*prefix*⟩`@`⟨*feature*⟩`$`⟨*sub*⟩.

```
\def\lst@IfRequired[#1]#2{%
    \lst@NormedDef\lst@temp{[#1]#2}%
    \expandafter\lst@IfRequired@\lst@temp\relax}
\def\lst@IfRequired@[#1]#2\relax#3{%
    \lst@IfOneOf #2$#1\relax\lst@require
        {\lst@DeleteKeysIn@\lst@require#2$#1,\relax,%
         \global\expandafter\let
             \csname\@lst\lst@prefix @#2$#1\endcsname\@empty
         #3}}
```

`\lst@require`

```
\let\lst@require\@empty
```

`\lst@NoAlias` just defines `\lst@oalias` and `\lst@malias`.

```
\def\lst@NoAlias[#1]#2{%
    \lst@NormedDef\lst@oalias{#1}\lst@NormedDef\lst@malias{#2}}
```

`\lst@LAS`

```
\gdef\lst@LAS#1#2#3#4#5#6#7{%
    \lst@Require{#1}{#2}{#3}#4#5%
    #4#3%
    \@ifundefined{lst#2@\lst@malias$\lst@oalias}%
        {\PackageError{Listings}%
         {#1 \ifx\@empty\lst@oalias\else \lst@oalias\space of \fi
          \lst@malias\space undefined}%
         {The #1 is not loadable. \@ehc}}%
        {#6\csname\@lst#2@\lst@malias $\lst@oalias\endcsname #7}}
```

`\lst@RequireAspects`
`\lstloadaspects`

make use of the just developped definitions.

```
\def\lst@RequireAspects#1{%
    \lst@Require{aspect}{asp}{#1}\lst@NoAlias\lstaspectfiles}
\let\lstloadaspects\lst@RequireAspects
```

`\lstaspectfiles`

This macro is defined if and only if it's undefined yet.

```
\@ifundefined{lstaspectfiles}
    {\newcommand\lstaspectfiles{lstmisc0.sty,lstmisc.sty}}{}
```

`\lst@DefDriver`

Test the next character and reinsert the arguments.

```
\gdef\lst@DefDriver#1#2#3#4{%
    \@ifnextchar[{\lst@DefDriver@{#1}{#2}#3#4}%
                 {\lst@DefDriver@{#1}{#2}#3#4[]}}
```

We set `\lst@if` locally true if the item has been requested.

```
\gdef\lst@DefDriver@#1#2#3#4[#5]#6{%
    \def\lst@name{#1}\let\lst@if#4%
    \lst@NormedDef\lst@driver{\@lst#2@#6$#5}%
    \lst@IfRequired[#5]{#6}{\begingroup \lst@true}%
                           {\begingroup}%
    \lst@setcatcodes
    \@ifnextchar[{\lst@XDefDriver{#1}#3}{\lst@DefDriver@@#3}}
```

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.

```
\gdef\lst@DefDriver@@#1#2{%
    \lst@if
        \global\@namedef{\lst@driver}{#1{#2}}%
    \fi
    \endgroup
    \@ifnextchar[\lst@XXDefDriver\@empty}
```

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.

```
\gdef\lst@XXDefDriver[#1]{%
    \ifx\@empty#1\@empty\else
        \lst@if
            \lstloadaspects{#1}%
        \else
            \@ifundefined{\lst@driver}{}%
            {\xdef\lst@loadaspects{\lst@loadaspects,#1}}%
        \fi
    \fi}
```

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

```
\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`.

```
⟨!info⟩\let\lst@UserCommand\gdef
⟨info⟩\def\lst@UserCommand#1{\message{\string#1,}\gdef#1}
```

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

```
\newcommand*\lst@BeginAspect[2][]{%
    \def\lst@curraspect{#2}%
    \ifx \lst@curraspect\@empty
        \expandafter\lst@GobbleAspect
    \else
```

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

```
⟨!info⟩         \let\lst@next\@empty
⟨info⟩          \def\lst@next{%
⟨info⟩              \message{^^JDefine lst-aspect ‘#2’:}\lst@GetAllocs}%
        \lst@IfRequired[]{#2}%
            {\lst@RequireAspects{#1}%
             \lst@if\else \let\lst@next\lst@GobbleAspect \fi}%
            {\let\lst@next\lst@GobbleAspect}%
        \expandafter\lst@next
    \fi}
```

`\lst@EndAspect` finishes an aspect definition.

```
\def\lst@EndAspect{%
    \csname\@lst patch@\lst@curraspect\endcsname
⟨info⟩     \lst@ReportAllocs
    \let\lst@curraspect\@empty}
```

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

```
\long\def\lst@GobbleAspect#1\lst@EndAspect{\let\lst@curraspect\@empty}
```

`\lst@Key` The command simply defines the key. But we must take care of an optional parameter and the initialization argument `#2`.

```
\def\lst@Key#1#2{%
⟨info⟩     \message{#1,}%
    \@ifnextchar[{\lstKV@def{#1}{#2}}%
                {\def\lst@temp{\lst@Key@{#1}{#2}}
                 \afterassignment\lst@temp
                 \global\@namedef{KV@\@lst @#1}####1}}
```

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

```
\def\lstKV@def#1#2[#3]{%
```

```
    \global\@namedef{KV@\@lst @#1@default\expandafter}\expandafter
        {\csname KV@\@lst @#1\endcsname{#3}}%
    \def\lst@temp{\lst@Key@{#1}{#2}}\afterassignment\lst@temp
    \global\@namedef{KV@\@lst @#1}##1}
```

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

```
\def\lst@Key@#1#2{%
    \ifx\relax#2\@empty\else
        \begingroup \globaldefs\@ne
        \csname KV@\@lst @#1\endcsname{#2}%
        \endgroup
    \fi}
```

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

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

`\lst@AddToHook`
`\lst@AddToHookExe`
`\lst@AddToHookAtTop`

All use the same submacro.

```
\def\lst@AddToHook{\lst@ATH@\iffalse\lst@AddTo}
\def\lst@AddToHookExe{\lst@ATH@\iftrue\lst@AddTo}
\def\lst@AddToHookAtTop{\lst@ATH@\iffalse\lst@AddToAtTop}
```

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

```
\long\def\lst@ATH@#1#2#3#4{%
    \@ifundefined{\@lst hk@#3}{%
⟨info⟩          \message{^^Jnew hook ‘#3’,^^J}%
        \expandafter\gdef\csname\@lst hk@#3\endcsname{}}{}%
    \expandafter#2\csname\@lst hk@#3\endcsname{#4}%
    \def\lst@temp{#4}%
    #1% \iftrue|false
        \begingroup \globaldefs\@ne \lst@temp \endgroup
    \fi}
```

`\lst@AddTo` Note that the definition is global!

```
\long\def\lst@AddTo#1#2{%
    \expandafter\gdef\expandafter#1\expandafter{#1#2}}
```

`\lst@AddToAtTop` We need a couple of `\expandafter`s now. Simply note that we have

`\expandafter\gdef\expandafter#1\expandafter{\lst@temp` ⟨*contents of* `#1`⟩`}`

after the ‘first phase’ of expansion.

```
\def\lst@AddToAtTop#1#2{\def\lst@temp{#2}%
    \expandafter\expandafter\expandafter\gdef
    \expandafter\expandafter\expandafter#1%
    \expandafter\expandafter\expandafter{\expandafter\lst@temp#1}}
```

`\lst@lAddTo` A local version of `\lst@AddTo` . . .

```
\def\lst@lAddTo#1#2{\expandafter\def\expandafter#1\expandafter{#1#2}}
```

`\lst@Extend`
`\lst@lExtend`

. . . and here we expand the first token of the second argument first.

```
\def\lst@Extend#1#2{%
    \expandafter\lst@AddTo\expandafter#1\expandafter{#2}}
\def\lst@lExtend#1#2{%
    \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 paramater `#1` to the definition part of the key macro. The following commands may be used to analyse the value. Note that we need at least version 1.10 of the keyval package. Note also that the package removes a naming conflict with AMS classes—reported by Ralf Quast.

```
\RequirePackage{keyval}[1997/11/10]
```

`\lstKV@TwoArg` `\lstKV@ThreeArg` `\lstKV@FourArg` Define temporary macros and call with given arguments `#1`. We add empty arguments for the case that the user doesn't provide enough.

```
\def\lstKV@TwoArg#1#2{\gdef\@gtempa##1##2{#2}\@gtempa#1{}{}}
\def\lstKV@ThreeArg#1#2{\gdef\@gtempa##1##2##3{#2}\@gtempa#1{}{}{}}
\def\lstKV@FourArg#1#2{\gdef\@gtempa##1##2##3##4{#2}\@gtempa#1{}{}{}{}}
```

There's one question: What are the global definitions good for? `\lst@Key` might set `\globaldefs` to one and possibly calls this macro. That's the reason why we use global definitions here and below.

`\lstKV@OptArg` We define the temporary macro `\@gtempa` and insert default argument if necessary.

```
\def\lstKV@OptArg[#1]#2#3{%
    \gdef\@gtempa[##1]##2{#3}\lstKV@OptArg@{#1}#2\@}
\def\lstKV@OptArg@#1{\@ifnextchar[\lstKV@OptArg@@{\lstKV@OptArg@@[#1]}}
\def\lstKV@OptArg@@[#1]#2\@{\@gtempa[#1]{#2}}
```

`\lstKV@XOptArg` Here `#3` is already a definition with at least two parameters whose first is enclosed in brackets.

```
\def\lstKV@XOptArg[#1]#2#3{%
    \global\let\@gtempa#3\lstKV@OptArg@{#1}#2\@}
```

`\lstKV@CSTwoArg` Just define temporary macro and call it.

```
\def\lstKV@CSTwoArg#1#2{%
    \gdef\@gtempa##1,##2,##3\relax{#2}%
    \@gtempa#1,,\relax}
```

`\lstKV@SetIf` We simply test the lower case first character of `#1`.

```
\def\lstKV@SetIf#1{\lstKV@SetIf@#1\relax}
\def\lstKV@SetIf@#1#2\relax#3{\lowercase{%
    \expandafter\let\expandafter#3%
        \csname if\ifx #1t}true\else false\fi\endcsname}
```

`\lstKV@SwitchCases` is implemented as a substring test. The original version used an `&`, which produced a bug—see p. 67.

```
\def\lstKV@SwitchCases#1#2#3{%
    \def\lst@temp##1\\#1:##2\\##3##4\@nil{%
        \ifx\@empty##3%
```

```
            #3%
        \else
            ##2%
        \fi
    }%
    \lst@temp\\#2\\#1:\\\@empty\@nil}
```

`\lstset` Finally this main user interface macro. We change catcodes for reading the argument.

```
\lst@UserCommand\lstset{\begingroup \lst@setcatcodes \lstset@}
\def\lstset@#1{\endgroup \ifx\@empty#1\@empty\else\setkeys{lst}{#1}\fi}
```

`\lst@setcatcodes` contains all catcode changes for `\lstset`. The equal-sign has been added after a bug report by Bekir Karaoglu—babel's active equal sign clashes with keyval's usage. ``\catcode`\"=12\relax`` has been removed after a bug report by Heiko Bauke — hopefully this introduces no other bugs.

```
\def\lst@setcatcodes{\makeatletter \catcode`\==12\relax}
```

To do: Change more catcodes?

13.8 Internal modes

`\lst@NewMode` We simply use `\chardef` for a mode definition. The counter `\lst@mode` mainly keeps the current mode number. But it is also used to advance the number in the macro `\lst@newmode`—we don't waste another counter.

```
\def\lst@NewMode#1{%
    \ifx\@undefined#1%
        \lst@mode\lst@newmode\relax \advance\lst@mode\@ne
        \xdef\lst@newmode{\the\lst@mode}%
        \global\chardef#1=\lst@mode
        \lst@mode\lst@nomode
    \fi}
```

`\lst@mode`
`\lst@nomode` We allocate the counter and the first mode.

```
\newcount\lst@mode
\def\lst@newmode{\m@ne}% init
\lst@NewMode\lst@nomode % init (of \lst@mode :-)
```

`\lst@UseDynamicMode` For dynamic modes we must not use the counter `\lst@mode` (since possibly already valued). `\lst@dynamicmode` substitutes `\lst@newmode` and is a local definition here, …

```
\def\lst@UseDynamicMode{%
    \@tempcnta\lst@dynamicmode\relax \advance\@tempcnta\@ne
    \edef\lst@dynamicmode{\the\@tempcnta}%
    \expandafter\lst@Swap\expandafter{\expandafter{\lst@dynamicmode}}}
```

… initialized each listing with the current ‘value’ of `\lst@newmode`.

```
\lst@AddToHook{InitVars}{\let\lst@dynamicmode\lst@newmode}
```

`\lst@EnterMode` Each mode opens a group level, stores the mode number and execute mode specific tokens. Moreover we keep all these changes in mind (locally) and adjust internal variables if the user wants it.

```
\def\lst@EnterMode#1#2{%
```

```
    \bgroup \lst@mode=#1\relax #2%
    \lst@FontAdjust
    \lst@lAddTo\lst@entermodes{\lst@EnterMode{#1}{#2}}}
```

```
\lst@AddToHook{InitVars}{\let\lst@entermodes\@empty}
\let\lst@entermodes\@empty % init
```

The initialization has been added after a bug report from Herfried Karl Wagner.

`\lst@LeaveMode` We simply close the group and call `\lsthk@EndGroup` if and only if the current mode is not `\lst@nomode`.

```
\def\lst@LeaveMode{%
    \ifnum\lst@mode=\lst@nomode\else
        \egroup \expandafter\lsthk@EndGroup
    \fi}
```

```
\lst@AddToHook{EndGroup}{}% init
```

`\lst@InterruptModes` We put the current mode sequence on a stack and leave all modes.

```
\def\lst@InterruptModes{%
    \lst@Extend\lst@modestack{\expandafter{\lst@entermodes}}%
    \lst@LeaveAllModes}
```

```
\lst@AddToHook{InitVars}{\global\let\lst@modestack\@empty}
```

`\lst@ReenterModes` If the stack is not empty, we leave all modes and pop the topmost element (which is the last element of `\lst@modestack`).

```
\def\lst@ReenterModes{%
    \ifx\lst@modestack\@empty\else
        \lst@LeaveAllModes
        \global\let\@gtempa\lst@modestack
        \global\let\lst@modestack\@empty
        \expandafter\lst@ReenterModes@\@gtempa\relax
    \fi}
\def\lst@ReenterModes@#1#2{%
    \ifx\relax#2\@empty
```

If we've reached `\relax`, we've also found the last element: we execute `#1` and gobble `{#2}`=`{\relax}` after `\fi`.

```
        \gdef\@gtempa##1{#1}%
        \expandafter\@gtempa
    \else
```

Otherwise we just add the element to `\lst@modestack` and continue the loop.

```
        \lst@AddTo\lst@modestack{{#1}}%
        \expandafter\lst@ReenterModes@
    \fi
    {#2}}
```

`\lst@LeaveAllModes` Leaving all modes means closing groups until the mode equals `\lst@nomode`.

```
\def\lst@LeaveAllModes{%
    \ifnum\lst@mode=\lst@nomode
        \expandafter\lsthk@EndGroup
    \else
        \expandafter\egroup\expandafter\lst@LeaveAllModes
    \fi}
```

We need that macro to end a listing correctly.

```
\lst@AddToHook{ExitVars}{\lst@LeaveAllModes}
```

`\lst@Pmode` `\lst@GPmode` The ‘processing’ and the general purpose mode.

```
\lst@NewMode\lst@Pmode
\lst@NewMode\lst@GPmode
```

`\lst@modetrue` The usual macro to value a boolean except that we also execute a hook.

```
\def\lst@modetrue{\let\lst@ifmode\iftrue \lsthk@ModeTrue}
\let\lst@ifmode\iffalse % init
\lst@AddToHook{ModeTrue}{}% init
```

`\lst@ifLmode` Comment lines use a static mode. It terminates at end of line.

```
\def\lst@Lmodetrue{\let\lst@ifLmode\iftrue}
\let\lst@ifLmode\iffalse % init
\lst@AddToHook{EOL}{\@whilesw \lst@ifLmode\fi \lst@LeaveMode}
```

13.9 Diverse helpers

`\lst@NormedDef` works like `\def` (without any parameters!) but normalizes the replacement text by making all characters lower case and stripping off spaces.

```
\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.

```
\def\lst@NormedNameDef#1#2{%
    \lowercase{\edef\lst@temp{\zap@space#1 \@empty}%
    \expandafter\xdef\csname\lst@temp\endcsname{\zap@space#2 \@empty}}}
```

`\lst@GetFreeMacro` Initialize `\@tempcnta` and `\lst@freemacro`, . . .

```
\def\lst@GetFreeMacro#1{%
    \@tempcnta\z@ \def\lst@freemacro{#1\the\@tempcnta}%
    \lst@GFM@}
```

. . . and either build the control sequence or advance the counter and continue.

```
\def\lst@GFM@{%
    \expandafter\ifx \csname\lst@freemacro\endcsname \relax
        \edef\lst@freemacro{\csname\lst@freemacro\endcsname}%
    \else
        \advance\@tempcnta\@ne
        \expandafter\lst@GFM@
    \fi}
```

`\lst@gtempboxa`

```
\newbox\lst@gtempboxa
```

```
⟨/kernel⟩
```

14 Doing output

14.1 Basic registers and keys

⟨∗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` `\lst@length` are allocated here. Quite a useful comment, isn't it?

```
\newtoks\lst@token \newcount\lst@length
```

`\lst@ResetToken` `\lst@lastother` The two registers get empty respectively zero at the beginning of each line. After receiving a report from Claus Atzenbeck—I removed such a bug many times—I decided to reset these registers in the `EndGroup` hook, too.

```
\def\lst@ResetToken{\lst@token{}\lst@length\z@}
```

```
\lst@AddToHook{InitVarsBOL}{\lst@ResetToken \let\lst@lastother\@empty}
\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.

```
\def\lst@lettertrue{\let\lst@ifletter\iftrue}
\def\lst@letterfalse{\let\lst@ifletter\iffalse}
\lst@AddToHook{InitVars}{\lst@letterfalse}
```

`\lst@Append` puts the argument into the output queue.

```
\def\lst@Append#1{\advance\lst@length\@ne
                  \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.

```
\def\lst@AppendOther{%
    \lst@ifletter \lst@Output\lst@letterfalse \fi
    \futurelet\lst@lastother\lst@Append}
```

`\lst@AppendLetter` We output a non-identifier string if necessary and call `\lst@Append`.

```
\def\lst@AppendLetter{%
    \lst@ifletter\else \lst@OutputOther\lst@lettertrue \fi
    \lst@Append}
```

`\lst@SaveToken` `\lst@RestoreToken` If a group end appears and ruins the character string, we can use these macros to save and restore the contents. `\lst@thestyle` is the current printing style and must be saved and restored, too.

```
\def\lst@SaveToken{%
    \global\let\lst@gthestyle\lst@thestyle
    \global\let\lst@glastother\lst@lastother
    \xdef\lst@RestoreToken{\noexpand\lst@token{\the\lst@token}%
                          \noexpand\lst@length\the\lst@length\relax
                          \noexpand\let\noexpand\lst@thestyle
                                       \noexpand\lst@gthestyle
                          \noexpand\let\noexpand\lst@lastother
                                       \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.

```
\def\lst@IfLastOtherOneOf#1{\lst@IfLastOtherOneOf@ #1\relax}
\def\lst@IfLastOtherOneOf@#1{%
    \ifx #1\relax
        \expandafter\@secondoftwo
    \else
        \ifx\lst@lastother#1%
            \lst@IfLastOtherOneOf@t
        \else
            \expandafter\expandafter\expandafter\lst@IfLastOtherOneOf@
        \fi
    \fi}
\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`
`\lst@column`
`\lst@pos`

the current line width and two counters.

```
\newdimen\lst@currlwidth % \global
\newcount\lst@column \newcount\lst@pos % \global
\lst@AddToHook{InitVarsBOL}
    {\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.

```
\def\lst@CalcColumn{%
            \@tempcnta\lst@column
    \advance\@tempcnta\lst@length
    \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.

```
\newdimen\lst@lostspace % \global
\lst@AddToHook{InitVarsBOL}{\global\lst@lostspace\z@}
```

`\lst@UseLostSpace` We insert space and reset it if and only if `\lst@lostspace` is positive.

```
\def\lst@UseLostSpace{\ifdim\lst@lostspace>\z@ \lst@InsertLostSpace \fi}
```

`\lst@InsertLostSpace`
`\lst@InsertHalfLostSpace`

Ditto, but insert even if negative. `\lst@Kern` will be defined very soon.

```
\def\lst@InsertLostSpace{%
    \lst@Kern\lst@lostspace \global\lst@lostspace\z@}
\def\lst@InsertHalfLostSpace{%
    \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`

`basewidth` assigns the values to macros and tests whether they are negative.

```
\newdimen\lst@width
\lst@Key{basewidth}{0.6em,0.45em}{\lstKV@CSTwoArg{#1}%
    {\def\lst@widthfixed{##1}\def\lst@widthflexible{##2}%
     \ifx\lst@widthflexible\@empty
         \let\lst@widthflexible\lst@widthfixed
     \fi
     \def\lst@temp{\PackageError{Listings}%
                                 {Negative value(s) treated as zero}%
                                 \@ehc}%
     \let\lst@error\@empty
     \ifdim \lst@widthfixed<\z@
         \let\lst@error\lst@temp \let\lst@widthfixed\z@
     \fi
     \ifdim \lst@widthflexible<\z@
         \let\lst@error\lst@temp \let\lst@widthflexible\z@
     \fi
     \lst@error}}
```

We set the dimension in a special hook.

```
\lst@AddToHook{FontAdjust}
    {\lst@width=\lst@ifflexible\lst@widthflexible
                               \else\lst@widthfixed\fi \relax}
```

`fontadjust` `\lst@FontAdjust`

This hook is controlled by a switch and is always executed at `InitVars`.

```
\lst@Key{fontadjust}{false}[t]{\lstKV@SetIf{#1}\lst@iffontadjust}
\def\lst@FontAdjust{\lst@iffontadjust \lsthk@FontAdjust \fi}
```

```
\lst@AddToHook{InitVars}{\lsthk@FontAdjust}
```

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` `\lst@alloverstyle`

The lowest level is the output of a box register. Here we use `\box#1` as argument to `\lst@alloverstyle`.

```
\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`.

```
\def\lst@alloverstyle#1{#1}% init
```

`\lst@Kern` has been used to insert 'lost space'. It must not use `\@tempboxa` since that ...

```
\def\lst@Kern#1{%
    \setbox\z@\hbox{{\lst@currstyle{\kern#1}}}%
    \global\advance\lst@currlwidth \wd\z@
    \lst@OutputBox\z@}
```

`\lst@CalcLostSpaceAndOutput` . . . is used here. We keep track of `\lst@lostspace`, `\lst@currlwidth` and `\lst@pos`.

```
\def\lst@CalcLostSpaceAndOutput{%
    \global\advance\lst@lostspace \lst@length\lst@width
    \global\advance\lst@lostspace-\wd\@tempboxa
    \global\advance\lst@currlwidth \wd\@tempboxa
    \global\advance\lst@pos -\lst@length
```

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`.

```
    \setbox\@tempboxa\hbox{\let\lst@OutputBox\box
        \ifdim\lst@lostspace>\z@ \lst@leftinsert \fi
        \box\@tempboxa
        \ifdim\lst@lostspace>\z@ \lst@rightinsert \fi}%
```

Finally we can output the new box.

```
    \lst@OutputBox\@tempboxa \lsthk@PostOutput}
```

```
\lst@AddToHook{PostOutput}{}% init
```

`\lst@OutputToken` Now comes a mid-level definition. Here we use `\lst@token` to set `\@tempboxa` and eventually output the box. We take care of font adjustment and special output styles. Yet unknown macros are defined in the following subsections.

```
\def\lst@OutputToken{%
    \lst@TrackNewLines \lst@OutputLostSpace
    \lst@ifgobbledws
        \lst@gobbledwhitespacefalse
        \lst@@discretionary
    \fi
    \lst@CheckMerge
    {\lst@thestyle{\lst@FontAdjust
     \setbox\@tempboxa\lst@hbox
        {\lsthk@OutputBox
         \lst@lefthss
         \expandafter\lst@FillOutputBox\the\lst@token\@empty
         \lst@righthss}%
     \lst@CalcLostSpaceAndOutput}}%
    \lst@ResetToken}
```

```
\lst@AddToHook{OutputBox}{}% init
```

```
\def\lst@gobbledwhitespacetrue{\global\let\lst@ifgobbledws\iftrue}
\def\lst@gobbledwhitespacefalse{\global\let\lst@ifgobbledws\iffalse}
\lst@AddToHookExe{InitBOL}{\lst@gobbledwhitespacefalse}% init
```

Delaying the output means saving the character string somewhere and pushing it back when neccessary. We may also attach the string to the next output box without affecting style detection: both will be printed in the style of the upcoming output. We will call this 'merging'.

`\lst@Delay`
`\lst@Merge` To delay or merge `#1`, we process it as usual and simply save the state in macros. For delayed characters we also need the currently 'active' output routine. Both definitions first check whether there are already delayed or 'merged' characters.

```
\def\lst@Delay#1{%
    \lst@CheckDelay
    #1%
    \lst@GetOutputMacro\lst@delayedoutput
    \edef\lst@delayed{\the\lst@token}%
    \edef\lst@delayedlength{\the\lst@length}%
    \lst@ResetToken}
```

```
\def\lst@Merge#1{%
    \lst@CheckMerge
    #1%
    \edef\lst@merged{\the\lst@token}%
    \edef\lst@mergedlength{\the\lst@length}%
    \lst@ResetToken}
```

`\lst@MergeToken` Here we put the things together again.

```
\def\lst@MergeToken#1#2{%
    \advance\lst@length#2%
    \lst@lExtend#1{\the\lst@token}%
    \expandafter\lst@token\expandafter{#1}%
    \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.

```
\def\lst@CheckDelay{%
    \ifx\lst@delayed\@empty\else
        \lst@GetOutputMacro\@gtempa
        \ifx\lst@delayedoutput\@gtempa
            \lst@MergeToken\lst@delayed\lst@delayedlength
        \else
            {\lst@ResetToken
             \lst@MergeToken\lst@delayed\lst@delayedlength
             \lst@delayedoutput}%
            \let\lst@delayed\@empty
        \fi
    \fi}
```

`\lst@CheckMerge` All this is easier for `\lst@merged`.

```
\def\lst@CheckMerge{%
    \ifx\lst@merged\@empty\else
        \lst@MergeToken\lst@merged\lst@mergedlength
    \fi}
```

```
\let\lst@delayed\@empty % init
\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.

```
\def\lst@column@fixed{%
    \lst@flexiblefalse
    \lst@width\lst@widthfixed\relax
    \let\lst@OutputLostSpace\lst@UseLostSpace
    \let\lst@FillOutputBox\lst@FillFixed
    \let\lst@hss\hss
    \def\lst@hbox{\hbox to\lst@length\lst@width}}
```

`\lst@FillFixed` Filling up a fixed mode box is easy.

```
\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.

```
\def\lst@FillFixed@#1{%
    \ifx\@empty#1\else \lst@hss#1\expandafter\lst@FillFixed@ \fi}
```

`\lst@column@flexible` The first flexible format.

```
\def\lst@column@flexible{%
    \lst@flexibletrue
    \lst@width\lst@widthflexible\relax
    \let\lst@OutputLostSpace\lst@UseLostSpace
    \let\lst@FillOutputBox\@empty
    \let\lst@hss\@empty
    \let\lst@hbox\hbox}
```

`\lst@column@fullflexible` This column format inserts no lost space except at the beginning of a line.

```
\def\lst@column@fullflexible{%
    \lst@column@flexible
    \def\lst@OutputLostSpace{\lst@ifnewline \lst@UseLostSpace\fi}%
    \let\lst@leftinsert\@empty
    \let\lst@rightinsert\@empty}
```

`\lst@column@spaceflexible` This column format only inserts lost space by stretching (invisible) existing spaces; it does not insert lost space between identifiers and other characters where the original does not have a space. It was suggested by Andrei Alexandrescu.

```
\def\lst@column@spaceflexible{%
    \lst@column@flexible
    \def\lst@OutputLostSpace{%
      \lst@ifwhitespace
        \ifx\lst@outputspace\lst@visiblespace
        \else
          \lst@UseLostSpace
        \fi
      \else
        \lst@ifnewline \lst@UseLostSpace\fi
      \fi}%
    \let\lst@leftinsert\@empty
    \let\lst@rightinsert\@empty}
```

Thus, we have the column formats. Now we define macros to use them.

`\lst@outputpos` This macro sets the ‘output-box-positioning’ parameter (the old key `outputpos`). We test for `l`, `c` and `r`. The fixed formats use `\lst@lefthss` and `\lst@righthss`, whereas the flexibles need `\lst@leftinsert` and `\lst@rightinsert`.

```
\def\lst@outputpos#1#2\relax{%
    \def\lst@lefthss{\lst@hss}\let\lst@righthss\lst@lefthss
    \let\lst@rightinsert\lst@InsertLostSpace
    \ifx #1c%
        \let\lst@leftinsert\lst@InsertHalfLostSpace
    \else\ifx #1r%
        \let\lst@righthss\@empty
        \let\lst@leftinsert\lst@InsertLostSpace
        \let\lst@rightinsert\@empty
    \else
        \let\lst@lefthss\@empty
        \let\lst@leftinsert\@empty
        \ifx #1l\else \PackageWarning{Listings}%
            {Unknown positioning for output boxes}%
        \fi
    \fi\fi}
```

`\lst@ifflexible` indicates the column mode but does not distinguish between different fixed or flexible modes.

```
\def\lst@flexibletrue{\let\lst@ifflexible\iftrue}
\def\lst@flexiblefalse{\let\lst@ifflexible\iffalse}
```

`columns` This is done here: check optional parameter and then build the control sequence of the column format.

```
\lst@Key{columns}{[c]fixed}{\lstKV@OptArg[]{#1}{%
    \ifx\@empty##1\@empty\else \lst@outputpos##1\relax\relax \fi
    \expandafter\let\expandafter\lst@arg
                                \csname\@lst @column@##2\endcsname
```

We issue a warning or save the definition for later.

```
    \lst@arg
    \ifx\lst@arg\relax
        \PackageWarning{Listings}{Unknown column format ‘##2’}%
    \else
        \lst@ifflexible
            \let\lst@columnsflexible\lst@arg
        \else
            \let\lst@columnsfixed\lst@arg
        \fi
    \fi}}
```

```
\let\lst@columnsfixed\lst@column@fixed % init
\let\lst@columnsflexible\lst@column@flexible % init
```

`flexiblecolumns` Nothing else but a key to switch between the last flexible and fixed mode.

```
\lst@Key{flexiblecolumns}\relax[t]{%
    \lstKV@SetIf{#1}\lst@ifflexible
    \lst@ifflexible \lst@columnsflexible
              \else \lst@columnsfixed \fi}
```

14.4 New lines

`\lst@newlines` This counter holds the number of 'new lines' (cr+lf) we have to perform.

```
\newcount\lst@newlines
\lst@AddToHook{InitVars}{\global\lst@newlines\z@}
\lst@AddToHook{InitVarsBOL}{\global\advance\lst@newlines\@ne}
```

`\lst@NewLine` This is how we start a new line: begin new paragraph and output an empty box. If low-level definition `\lst@OutputBox` just gobbles the box , we don't start a new line. This is used to drop the whole output.

```
\def\lst@NewLine{%
    \ifx\lst@OutputBox\@gobble\else
        \par\noindent \hbox{}%
    \fi
    \global\advance\lst@newlines\m@ne
    \lst@newlinetrue}
```

Define `\lst@newlinetrue` and reset if after output.

```
\def\lst@newlinetrue{\global\let\lst@ifnewline\iftrue}
\lst@AddToHookExe{PostOutput}{\global\let\lst@ifnewline\iffalse}% init
```

`\lst@TrackNewLines` If `\lst@newlines` is positive, we execute the hook and insert the new lines.

```
\def\lst@TrackNewLines{%
    \ifnum\lst@newlines>\z@
        \lsthk@OnNewLine
        \lst@DoNewLines
    \fi}
\lst@AddToHook{OnNewLine}{}% init
```

`emptylines` Adam Prugel-Bennett asked for such a key—if I didn't misunderstood him. We check for the optional star and set `\lst@maxempty` and switch.

```
\lst@Key{emptylines}\maxdimen{%
    \@ifstar{\lst@true\@tempcnta\@gobble#1\relax\lst@GobbleNil}%
            {\lst@false\@tempcnta#1\relax\lst@GobbleNil}#1\@nil
    \advance\@tempcnta\@ne
    \edef\lst@maxempty{\the\@tempcnta\relax}%
    \let\lst@ifpreservenumber\lst@if}
```

`\lst@DoNewLines` First we take care of `\lst@maxempty` and then of the remaining empty lines.

```
\def\lst@DoNewLines{
    \@whilenum\lst@newlines>\lst@maxempty \do
        {\lst@ifpreservenumber
            \lsthk@OnEmptyLine
            \global\advance\c@lstnumber\lst@advancelstnum
         \fi
         \global\advance\lst@newlines\m@ne}%
    \@whilenum \lst@newlines>\@ne \do
        {\lsthk@OnEmptyLine \lst@NewLine}%
    \ifnum\lst@newlines>\z@ \lst@NewLine \fi}
\lst@AddToHook{OnEmptyLine}{}% init
```

14.5 High-level output

`identifierstyle` A simple key.

```
\lst@Key{identifierstyle}{}{\def\lst@identifierstyle{#1}}
\lst@AddToHook{EmptyStyle}{\let\lst@identifierstyle\@empty}
```

`\lst@GotoTabStop` Here we look whether the line already contains printed characters. If true, we output a box with the width of a blank space.

```
\def\lst@GotoTabStop{%
    \ifnum\lst@newlines=\z@
        \setbox\@tempboxa\hbox{\lst@outputspace}%
        \setbox\@tempboxa\hbox to\wd\@tempboxa{{\lst@currstyle{\hss}}}%
        \lst@CalcLostSpaceAndOutput
```

It's probably not clear why it is sufficient to output a single space to go to the next tabulator stop. Just note that the space lost by this process is 'lost space' in the sense above and therefore will be inserted before the next characters are output.

```
    \else
```

Otherwise (no printed characters) we only need to advance `\lst@lostspace`, which is inserted by `\lst@OutputToken` above, and update the column.

```
        \global\advance\lst@lostspace \lst@length\lst@width
        \global\advance\lst@column\lst@length \lst@length\z@
    \fi}
```

Note that this version works also in flexible column mode. In fact, it's mainly the flexible version of listings 0.20.

To do: Use `\lst@ifnewline` instead of `\ifnum\lst@newlines=\z@`?

`\lst@OutputOther` becomes easy with the previous definitions.

```
\def\lst@OutputOther{%
    \lst@CheckDelay
    \ifnum\lst@length=\z@\else
        \let\lst@thestyle\lst@currstyle
        \lsthk@OutputOther
        \lst@OutputToken
    \fi}
```

```
\lst@AddToHook{OutputOther}{}% init
\let\lst@currstyle\relax % init
```

`\lst@Output` We might use identifier style as default.

```
\def\lst@Output{%
    \lst@CheckDelay
    \ifnum\lst@length=\z@\else
        \ifx\lst@currstyle\relax
            \let\lst@thestyle\lst@identifierstyle
        \else
            \let\lst@thestyle\lst@currstyle
        \fi
        \lsthk@Output
        \lst@OutputToken
    \fi
    \let\lst@lastother\relax}
```

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.

```
\lst@AddToHook{Output}{}% init
```

`\lst@GetOutputMacro` Just saves the output macro to be used.

```
\def\lst@GetOutputMacro#1{%
    \lst@ifletter \global\let#1\lst@Output
            \else \global\let#1\lst@OutputOther\fi}
```

`\lst@PrintToken` outputs the current character string in letter or nonletter mode.

```
\def\lst@PrintToken{%
    \lst@ifletter \lst@Output \lst@letterfalse
            \else \lst@OutputOther \let\lst@lastother\@empty \fi}
```

`\lst@XPrintToken` is a special definition to print also merged characters.

```
\def\lst@XPrintToken{%
    \lst@PrintToken \lst@CheckMerge
    \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

```
\def\lst@BeginDropOutput#1{%
    \xdef\lst@BDOnewlines{\the\lst@newlines}%
    \global\let\lst@BDOifnewline\lst@ifnewline
    \lst@EnterMode{#1}%
        {\lst@modetrue
         \let\lst@OutputBox\@gobble
         \aftergroup\lst@BDORestore}}
```

Restoring the date is quite easy:

```
\def\lst@BDORestore{%
    \global\lst@newlines\lst@BDOnewlines
    \global\let\lst@ifnewline\lst@BDOifnewline}
```

`\lst@EndDropOutput` is equivalent to `\lst@LeaveMode`.

```
\let\lst@EndDropOutput\lst@LeaveMode
```

```
⟨/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.

```
⟨*misc⟩
\lst@BeginAspect{writefile}
```

`\lst@WF` `\lst@WFtoken` The contents of the token will be written to file.

```
\newtoks\lst@WFtoken % global
\lst@AddToHook{InitVarsBOL}{\global\lst@WFtoken{}}
```

```
\newwrite\lst@WF
\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.

```
\gdef\lst@WFWriteToFile{%
  \begingroup
   \let\lst@UM\@empty
   \expandafter\edef\expandafter\lst@temp\expandafter{\the\lst@WFtoken}%
   \immediate\write\lst@WF{\lst@temp}%
  \endgroup
  \global\lst@WFtoken{}}
```

`\lst@WFAppend` Similar to `\lst@Append` but uses `\lst@WFtoken`.

```
\gdef\lst@WFAppend#1{%
    \global\lst@WFtoken=\expandafter{\the\lst@WFtoken#1}}
```

`\lst@BeginWriteFile` `\lst@BeginAlsoWriteFile` use different macros for `\lst@OutputBox` (not) to drop the output.

```
\gdef\lst@BeginWriteFile{\lst@WFBegin\@gobble}
\gdef\lst@BeginAlsoWriteFile{\lst@WFBegin\lst@OutputBox}
```

`\lst@WFBegin` Here …

```
\begingroup \catcode`\^^I=11
\gdef\lst@WFBegin#1#2{%
    \begingroup
    \let\lst@OutputBox#1%
```

… we have to update `\lst@WFtoken` and …

```
    \def\lst@Append##1{%
        \advance\lst@length\@ne
        \expandafter\lst@token\expandafter{\the\lst@token##1}%
        \ifx ##1\lst@outputspace \else
            \lst@WFAppend##1%
        \fi}%
    \lst@lAddTo\lst@PreGotoTabStop{\lst@WFAppend{^^I}}%
    \lst@lAddTo\lst@ProcessSpace{\lst@WFAppend{ }}%
```

… need different 'EOL' and 'DeInit' definitions to write the token register to file.

```
    \let\lst@DeInit\lst@WFDeInit
    \let\lst@MProcessListing\lst@WFMProcessListing
```

Finally we open the file if necessary.

```
    \lst@WFifopen\else
        \immediate\openout\lst@WF=#2\relax
        \global\let\lst@WFifopen\iftrue
        \@gobbletwo\fi\fi
    \fi}
\endgroup
```

`\lst@EndWriteFile` closes the file and restores original definitions.

```
\gdef\lst@EndWriteFile{%
    \immediate\closeout\lst@WF \endgroup
    \global\let\lst@WFifopen\iffalse}
```

`\lst@WFMProcessListing` `\lst@WFDeInit` write additionally `\lst@WFtoken` to external file.

```
\global\let\lst@WFMProcessListing\lst@MProcessListing
\global\let\lst@WFDeInit\lst@DeInit
\lst@AddToAtTop\lst@WFMProcessListing{\lst@WFWriteToFile}
\lst@AddToAtTop\lst@WFDeInit{%
    \ifnum\lst@length=\z@\else \lst@WFWriteToFile \fi}

\lst@EndAspect
⟨/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

```
⟨*kernel⟩
```

`\lst@ProcessLetter` We put the letter, which is not a whitespace, into the output queue.

```
\def\lst@ProcessLetter{\lst@whitespacefalse \lst@AppendLetter}
```

`\lst@ProcessOther` Ditto.

```
\def\lst@ProcessOther{\lst@whitespacefalse \lst@AppendOther}
```

`\lst@ProcessDigit` A digit appends the character to the current character string. But we must use the right macro. This allows digits to be part of an identifier or a numerical constant.

```
\def\lst@ProcessDigit{%
    \lst@whitespacefalse
    \lst@ifletter \expandafter\lst@AppendLetter
            \else \expandafter\lst@AppendOther\fi}
```

`\lst@ifwhitespace` indicates whether the last processed character has been white space.

```
\def\lst@whitespacetrue{\global\let\lst@ifwhitespace\iftrue}
\def\lst@whitespacefalse{\global\let\lst@ifwhitespace\iffalse}
\lst@AddToHook{InitVarsBOL}{\lst@whitespacetrue}
```

15.2 Whitespaces

Here we have to take care of two things: dropping empty lines at the end of a listing and the different column formats. Both use `\lst@lostspace`. Lines containing only tabulators and spaces should be viewed as empty. In order to achieve this, tabulators and spaces at the beginning of a line don't output any characters but advance `\lst@lostspace`. Whenever this dimension is positive we insert that space before the character string is output. Thus, if there are only tabulators and spaces, the line is 'empty' since we haven't done any output.

We have to do more for flexible columns. Whitespaces can fix the column alignment: if the real line is wider than expected, a tabulator is at least one space wide; all remaining space fixes the alignment. If there are two or more space characters, at least one is printed; the others fix the column alignment.

Tabulators are processed in three stages. You have already seen the last stage `\lst@GotoTabStop`. The other two calculate the necessary width and take care of visible tabulators and spaces.

`tabsize` We check for a legal argument before saving it. Default tabsize is 8 as proposed by Rolf Niepraschk.

```
\lst@Key{tabsize}{8}
    {\ifnum#1>\z@ \def\lst@tabsize{#1}\else
         \PackageError{Listings}{Strict positive integer expected}%
         {You can’t use ‘#1’ as tabsize. \@ehc}%
     \fi}
```

`showtabs` `tab` Two more user keys for tab control.

```
\lst@Key{showtabs}f[t]{\lstKV@SetIf{#1}\lst@ifshowtabs}
\lst@Key{tab}{\kern.06em\hbox{\vrule\@height.3ex}%
              \hrulefill\hbox{\vrule\@height.3ex}}
    {\def\lst@tab{#1}}
```

`\lst@ProcessTabulator` A tabulator outputs the preceding characters, which decrements `\lst@pos` by the number of printed characters.

```
\def\lst@ProcessTabulator{%
    \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.

```
    \global\advance\lst@column -\lst@pos
    \@whilenum \lst@pos<\@ne \do
        {\global\advance\lst@pos\lst@tabsize}%
    \lst@length\lst@pos
    \lst@PreGotoTabStop}
```

`\lst@PreGotoTabStop` Visible tabs print `\lst@tab`.

```
\def\lst@PreGotoTabStop{%
    \lst@ifshowtabs
        \lst@TrackNewLines
        \setbox\@tempboxa\hbox to\lst@length\lst@width
            {{\lst@currstyle{\hss\lst@tab}}}%
        \lst@CalcLostSpaceAndOutput
    \else
```

If we are advised to keep spaces, we insert the correct number of them.

```
        \lst@ifkeepspaces
            \@tempcnta\lst@length \lst@length\z@
            \@whilenum \@tempcnta>\z@ \do
                {\lst@AppendOther\lst@outputspace
                 \advance\@tempcnta\m@ne}%
            \lst@OutputOther
        \else
            \lst@GotoTabStop
        \fi
    \fi
    \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` `\lst@visiblespace` Denis Bitouzé pointed out, that, with LuaLaTeX and some monospaced font which doesn't have an appropriate glyph in slot 32, 'showspaces' hasn't any effect by using `\textvisiblespace`. So now we're using `\verbvisiblespace` as a default definition for the first macro, the test for the fontfamily `\lst@ttfamily` is deleted, because in most of the cases it lead to the (wrong) output of `\char32`. The definition of `\verbvisible` from latex.ltx takes the different behaviour of the modern TeX engines into account and defines a valid `\verbvisiblespace`

```
\def\lst@outputspace{\ }
\def\lst@visiblespace{\verbvisiblespace}
```

`showspaces` `keepspaces` . . . which is modified on user's request.

```
\lst@Key{showspaces}{false}[t]{\lstKV@SetIf{#1}\lst@ifshowspaces}
\lst@Key{keepspaces}{false}[t]{\lstKV@SetIf{#1}\lst@ifkeepspaces}
\lst@AddToHook{Init}
    {\lst@ifshowspaces
         \let\lst@outputspace\lst@visiblespace
         \lst@keepspacestrue
     \fi}
\def\lst@keepspacestrue{\let\lst@ifkeepspaces\iftrue}
```

`\lst@ProcessSpace` We look whether spaces fix the column alignment or not. In the latter case we append a space; otherwise . . . Andrei Alexandrescu tested the `spaceflexible` column setting and found a bug that resulted from `\lst@PrintToken` and `\lst@whitespacetrue` being out of order here.

```
\def\lst@ProcessSpace{%
    \lst@ifkeepspaces
        \lst@PrintToken
        \lst@whitespacetrue
        \lst@AppendOther\lst@outputspace
        \lst@PrintToken
    \else \ifnum\lst@newlines=\z@
```

. . . we append a 'special space' if the line isn't empty.

```
        \lst@AppendSpecialSpace
    \else \ifnum\lst@length=\z@
```

If the line is empty, we check whether there are characters in the output queue. If there are no characters we just advance `\lst@lostspace`. Otherwise we append the space.

```
            \global\advance\lst@lostspace\lst@width
            \global\advance\lst@pos\m@ne
            \lst@whitespacetrue
        \else
            \lst@AppendSpecialSpace
        \fi
    \fi \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. Also, `\lst@whitespacetrue` has been moved after `\lst@PrintToken` so that the token-printer can correctly check whether it is printing whitespace or not; this was preventing the `spaceflexible` column setting from working correctly.

```
\def\lst@AppendSpecialSpace{%
    \lst@ifwhitespace
        \lst@PrintToken
        \global\advance\lst@lostspace\lst@width
        \global\advance\lst@pos\m@ne
        \lst@gobbledwhitespacetrue
    \else
        \lst@PrintToken
        \lst@whitespacetrue
        \lst@AppendOther\lst@outputspace
        \lst@PrintToken
    \fi}
```

Form feeds has been introduced after communication with Jan Braun.

`formfeed` let the user make adjustments.

```
\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.

```
\def\lst@ProcessFormFeed{%
    \lst@XPrintToken
    \ifnum\lst@newlines=\z@
        \lst@EOLUpdate \lsthk@InitVarsBOL
    \fi
    \lst@formfeed
    \lst@whitespacetrue}
```

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`⟨*class macro*⟩$\langle c_1\rangle \ldots \langle c_k\rangle$`\z@`

> extends the standard character table by the characters with codes $\langle c_1\rangle \ldots \langle c_k\rangle$ making each character use ⟨*class macro*⟩. All these characters must be printable via `\char`$\langle c_i\rangle$.

`\lst@CCPutMacro`$\langle \mathit{class}_1\rangle\langle c_1\rangle\langle \mathit{definition}_1\rangle\ldots$`\@empty\z@\@empty`

> also extends the standard character table: the character $\langle c_i\rangle$ will use $\langle \mathit{class}_i\rangle$ and is printed via $\langle \mathit{definition}_i\rangle$. These definitions must be ⟨*spec. token*⟩s in the sense of section 9.5.

`\lst@Def` `\lst@Let` For speed we won't use these helpers too often.

```
\def\lst@Def#1{\lccode`\~=#1\lowercase{\def~}}
\def\lst@Let#1{\lccode`\~=#1\lowercase{\let~}}
```

The definition of the space below doesn't hurt anything. But other aspects, for example lineshape and formats, redefine also the macro `\space`. Now, if LaTeX calls `\try@load@fontshape`, the `.log` messages would show some strange things since LaTeX uses `\space` in these messages. The following addition ensures that `\space` expands to a space and not to something different. This was one more bug reported by Denis Girou.

```
\lst@AddToAtTop{\try@load@fontshape}{\def\space{ }}
```

`\lst@SelectStdCharTable` The first three standard characters. `\lst@Let` has been replaced by `\lst@Def` after a bug report from Chris Edwards.

```
\def\lst@SelectStdCharTable{%
    \lst@Def{9}{\lst@ProcessTabulator}%
    \lst@Def{12}{\lst@ProcessFormFeed}%
    \lst@Def{32}{\lst@ProcessSpace}}
```

`\lst@CCPut` The first argument gives the character class, then follow the codes.

Joseph Wright pointed to a bug which came up on TeX StackExchange (`http://tex.stackexchange.com/questions/302437/textcase-lstings-and-tilde`). Other than in `\lst@CCPutMacro` the `\lccode` settings weren't local and caused the error.

```
\def\lst@CCPut#1#2{%
    \ifnum#2=\z@
        \expandafter\@gobbletwo
    \else
        \begingroup\lccode`\~=#2\lccode`\/=#2\lowercase{\endgroup\lst@CCPut@~{#1/}}%
    \fi
    \lst@CCPut#1}
\def\lst@CCPut@#1#2{\lst@lAddTo\lst@SelectStdCharTable{\def#1{#2}}}
```

Now we insert more standard characters.

```
\lst@CCPut \lst@ProcessOther
    {"21}{"22}{"28}{"29}{"2B}{"2C}{"2E}{"2F}
    {"3A}{"3B}{"3D}{"3F}{"5B}{"5D}
    \z@
\lst@CCPut \lst@ProcessDigit
    {"30}{"31}{"32}{"33}{"34}{"35}{"36}{"37}{"38}{"39}
    \z@
\lst@CCPut \lst@ProcessLetter
    {"40}{"41}{"42}{"43}{"44}{"45}{"46}{"47}
    {"48}{"49}{"4A}{"4B}{"4C}{"4D}{"4E}{"4F}
    {"50}{"51}{"52}{"53}{"54}{"55}{"56}{"57}
    {"58}{"59}{"5A}
         {"61}{"62}{"63}{"64}{"65}{"66}{"67}
    {"68}{"69}{"6A}{"6B}{"6C}{"6D}{"6E}{"6F}
    {"70}{"71}{"72}{"73}{"74}{"75}{"76}{"77}
    {"78}{"79}{"7A}
    \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.

```
\def\lst@CCPutMacro#1#2#3{%
    \ifnum#2=\z@ \else
        \begingroup\lccode`\~=#2\relax \lccode`\/=#2\relax
        \lowercase{\endgroup\expandafter\lst@CCPutMacro@
            \csname\@lst @um/\expandafter\endcsname
            \csname\@lst @um/@\endcsname /~}#1{#3}%
        \expandafter\lst@CCPutMacro
    \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_@`.

```
\def\lst@CCPutMacro@#1#2#3#4#5#6{%
    \lst@lAddTo\lst@SelectStdCharTable{\def#4{#5#1}}%
    \def#1{\lst@UM#3}%
    \def#2{#6}}
```

The default definition of `\lst@UM`:

```
\def\lst@UM#1{\csname\@lst @um#1@\endcsname}
```

And all remaining standard characters.

```
\lst@CCPutMacro
    \lst@ProcessOther {"23}\#
    \lst@ProcessLetter{"24}\textdollar
    \lst@ProcessOther {"25}\%
    \lst@ProcessOther {"26}\&
    \lst@ProcessOther {"27}{\lst@ifupquote \textquotesingle
                                        \else \char39\relax \fi}
    \lst@ProcessOther {"2A}{\lst@ttfamily*\textasteriskcentered}
```

Ulrike Fischer pointed out the incompatibility between flexisym and listings: flexisym changes the math code while listings changes the meaning. So the minus character vanishes. Replacing the original `$-$` by `\textminus` should remedy the problem.

```
    \lst@ProcessOther {"2D}{\lst@ttfamily{-{}}{\textminus}}
    \lst@ProcessOther {"3C}{\lst@ttfamily<\textless}
    \lst@ProcessOther {"3E}{\lst@ttfamily>\textgreater}
    \lst@ProcessOther {"5C}{\lst@ttfamily{\char92}\textbackslash}
    \lst@ProcessOther {"5E}\textasciicircum
    \lst@ProcessLetter{"5F}{\lst@ttfamily{\char95}\textunderscore}
    \lst@ProcessOther {"60}{\lst@ifupquote \textasciigrave
                                     \else \char96\relax \fi}
    \lst@ProcessOther {"7B}{\lst@ttfamily{\char123}\textbraceleft}
    \lst@ProcessOther {"7C}{\lst@ttfamily|\textbar}
    \lst@ProcessOther {"7D}{\lst@ttfamily{\char125}\textbraceright}
    \lst@ProcessOther {"7E}\textasciitilde
    \lst@ProcessOther {"7F}-
    \@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

```
\def\lst@ttfamily#1#2{\ifx\f@family\ttdefault#1\relax\else#2\fi}
```

`\ttdefault` is defined `\long`, so the `\ifx` doesn't work since `\f@family` isn't `\long`! We go around this problem by redefining `\ttdefault` locally:

```
\lst@AddToHook{Init}{\edef\ttdefault{\ttdefault}}
```

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.

```
\lst@Key{upquote}{false}[t]{\lstKV@SetIf{#1}\lst@ifupquote
    \lst@ifupquote
       \@ifundefined{textasciigrave}%
          {\let\KV@lst@upquote\@gobble
           \lstKV@SetIf f\lst@ifupquote \@gobble\fi
           \PackageError{Listings}{Option ‘upquote’ requires ‘textcomp’
            package.\MessageBreak The option has been disabled}%
          {Add \string\usepackage{textcomp} to your preamble.}}%
          {}%
    \fi}
```

If an upquote package is loaded, the upquote option is enabled by default.

```
\AtBeginDocument{%
 \@ifpackageloaded{upquote}{\RequirePackage{textcomp}%
                             \lstset{upquote}}{}%
 \@ifpackageloaded{upquote2}{\lstset{upquote}}{}}
```

`\lst@ifactivechars` A simple switch.

```
\def\lst@activecharstrue{\let\lst@ifactivechars\iftrue}
\def\lst@activecharsfalse{\let\lst@ifactivechars\iffalse}
\lst@activecharstrue
```

`\lst@SelectCharTable` We select the standard character table and switch to active catcodes.

```
\def\lst@SelectCharTable{%
    \lst@SelectStdCharTable
    \lst@ifactivechars
        \catcode9\active \catcode12\active \catcode13\active
        \@tempcnta=32\relax
        \@whilenum\@tempcnta<128\do
            {\catcode\@tempcnta\active\advance\@tempcnta\@ne}%
    \fi
    \lst@ifec \lst@DefEC \fi
```

The following line and the according macros below have been added after a bug report from Frédéric Boulanger. The assignment to `\do@noligs` was changed to `\do` after a bug report from Peter Ruckdeschel. This bugfix was kindly provided by Timothy Van Zandt.

```
    \let\do\lst@do@noligs \verbatim@nolig@list
```

There are two ways to adjust the standard table: inside the hook or with `\lst@DeveloperSCT`. We use these macros and initialize the backslash if necessary. `\lst@DefRange` has been moved outside the hook after a bug report by Michael Bachmann.

```
    \lsthk@SelectCharTable
    \lst@DeveloperSCT
    \lst@DefRange
    \ifx\lst@Backslash\relax\else
        \lst@LetSaveDef{"5C}\lsts@backslash\lst@Backslash
    \fi}
```

`SelectCharTable`
`MoreSelectCharTable`

The keys to adjust `\lst@DeveloperSCT`.

```
\lst@Key{SelectCharTable}{}{\def\lst@DeveloperSCT{#1}}
\lst@Key{MoreSelectCharTable}\relax{\lst@lAddTo\lst@DeveloperSCT{#1}}
```

```
\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.

```
\def\lst@do@noligs#1{%
    \begingroup \lccode`\~=`#1\lowercase{\endgroup
    \lst@do@noligs@~}}
\def\lst@do@noligs@#1{%
    \expandafter\expandafter\expandafter\def
    \expandafter\expandafter\expandafter#1%
    \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.

```
\def\lst@NoLig{\advance\lst@length\m@ne \lst@Append\lst@nolig}
\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 LaTeX's `\do@noligs`.

```
\@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.

```
\def\lst@SaveOutputDef#1#2{%
    \begingroup \lccode‘\~=#1\relax \lowercase{\endgroup
    \def\lst@temp##1\def~##2##3\relax}{%
        \global\expandafter\let\expandafter#2\@gobble##2\relax}%
    \expandafter\lst@temp\lst@SelectStdCharTable\relax}
```

`\lstum@backslash` A commonly used character.

```
\lst@SaveOutputDef{"5C}\lstum@backslash
```

15.3.2 National characters

`extendedchars` The user key to activate extended characters 128–255.

```
\lst@Key{extendedchars}{true}[t]{\lstKV@SetIf{#1}\lst@ifec}
```

`\lst@DefEC` Currently each character in the range 128–255 is treated as a letter.

```
\def\lst@DefEC{%
    \lst@CCECUse \lst@ProcessLetter
      ^^80^^81^^82^^83^^84^^85^^86^^87^^88^^89^^8a^^8b^^8c^^8d^^8e^^8f%
      ^^90^^91^^92^^93^^94^^95^^96^^97^^98^^99^^9a^^9b^^9c^^9d^^9e^^9f%
      ^^a0^^a1^^a2^^a3^^a4^^a5^^a6^^a7^^a8^^a9^^aa^^ab^^ac^^ad^^ae^^af%
      ^^b0^^b1^^b2^^b3^^b4^^b5^^b6^^b7^^b8^^b9^^ba^^bb^^bc^^bd^^be^^bf%
      ^^c0^^c1^^c2^^c3^^c4^^c5^^c6^^c7^^c8^^c9^^ca^^cb^^cc^^cd^^ce^^cf%
      ^^d0^^d1^^d2^^d3^^d4^^d5^^d6^^d7^^d8^^d9^^da^^db^^dc^^dd^^de^^df%
      ^^e0^^e1^^e2^^e3^^e4^^e5^^e6^^e7^^e8^^e9^^ea^^eb^^ec^^ed^^ee^^ef%
      ^^f0^^f1^^f2^^f3^^f4^^f5^^f6^^f7^^f8^^f9^^fa^^fb^^fc^^fd^^fe^^ff%
      ^^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.

```
\def\lst@CCECUse#1#2{%
    \ifnum‘#2=\z@
        \expandafter\@gobbletwo
    \else
        \ifnum\catcode‘#2=\active
            \lccode‘\~=‘#2\lccode‘\/=‘#2\lowercase{\lst@CCECUse@#1~/}%
        \else
            \lst@ifactivechars \catcode‘#2=\active \fi
            \lccode‘\~=‘#2\lccode‘\/=‘#2\lowercase{\def~{#1/}}%
        \fi
    \fi
    \lst@CCECUse#1}
```

We save the meaning as mentioned. Here we must also use the ‘`\lst@UM` construction’ since extended characters could often appear in words = identifiers. Bug reported by Denis Girou.

```
\def\lst@CCECUse@#1#2#3{%
    \expandafter\def\csname\@lst @EC#3\endcsname{\lst@UM#3}%
    \expandafter\let\csname\@lst @um#3@\endcsname #2%
    \edef#2{\noexpand#1%
            \expandafter\noexpand\csname\@lst @EC#3\endcsname}}
```

Daniel Gerigk and Heiko Oberdiek reported an error and a solution, respectively.

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`.

```
\lst@AddToHook{Init}
    {\let\lsts@nfss@catcodes\nfss@catcodes
     \let\nfss@catcodes\lst@nfss@catcodes}
```

The &-character had turned into `\&` after a bug report by David Aspinall.

```
\def\lst@nfss@catcodes{%
    \lst@makeletter
        ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz\relax
    \@makeother (\@makeother )\@makeother ,\@makeother :\@makeother\&%
    \@makeother 0\@makeother 1\@makeother 2\@makeother 3\@makeother 4%
    \@makeother 5\@makeother 6\@makeother 7\@makeother 8\@makeother 9%
    \@makeother =\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 `\@makeaother (`.

The name of `\lst@makeletter` is an imitation of LaTeX’s `\@makeother`.

```
\def\lst@makeletter#1{%
    \ifx\relax#1\else\catcode`#111\relax \expandafter\lst@makeletter\fi}
```

`useoutput`
`\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 TeX seems to use the *currently* active catcode table when it writes non-`\immediate` `\write`s 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.

We can choose between three possibilities. Donald Arseneau noted a bug here in the `\ifcase` argument.

```
\lst@Key{useoutput}{2}{\edef\lst@useoutput{\ifcase0#1 0\or 1\else 2\fi}}
```

The first does not modify the existing output routine.

```
\lst@AddToHook{Init}
{\edef\lst@OrgOutput{\the\output}%
\ifcase\lst@useoutput\relax
\or
```

The second possibility is as follows: We interrupt the current modes—in particular `\lst@Pmode` with modified catcode table—, call the original output routine and reenter the mode. This must be done with a little care. First we have to close the group which TeX opens at the beginning of the output routine. A single `\egroup` gives an ‘unbalanced output routine’ error. But `\expandafter\egroup` works. Again it was Donald Arseneau who gave the explaination: 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 temporaryly.

```
 \output{\global\setbox\lst@gtempboxa\box\@cclv
         \expandafter\egroup
```

Now we can interrupt the mode, but we have to save the current character string and the current style.

```
         \lst@SaveToken
     \lst@InterruptModes
```

We restore the contents, use the original output routine, and ...

```
     \setbox\@cclv\box\lst@gtempboxa
     \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.

```
     \bgroup
     \aftergroup\pagegoal\aftergroup\vsize
     \aftergroup\lst@ReenterModes\aftergroup\lst@RestoreToken}%
\else
```

The third option is to restore all catcodes and meanings inside a modified output routine and to call the original routine afterwards.

```
 \output{\lst@RestoreOrigCatcodes
         \lst@ifec \lst@RestoreOrigExtendedCatcodes \fi
         \lst@OrgOutput}%
\fi}
```

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.

`\lst@GetChars`
`\lst@ScanChars`
`rescanchars`

To make the third `\output`-option work, we have to scan the catcodes and also the meanings of active characters:

```
\def\lst@GetChars#1#2#3{%
    \let#1\@empty
    \@tempcnta#2\relax \@tempcntb#3\relax
    \loop \ifnum\@tempcnta<\@tempcntb\relax
        \lst@lExtend#1{\expandafter\catcode\the\@tempcnta=}%
        \lst@lExtend#1{\the\catcode\@tempcnta\relax}%
        \ifnum\the\catcode\@tempcnta=\active
            \begingroup\lccode`\~=\@tempcnta
            \lowercase{\endgroup
            \lst@lExtend#1{\expandafter\let\expandafter~\csname
                                    lstecs@\the\@tempcnta\endcsname}%
            \expandafter\let\csname lstecs@\the\@tempcnta\endcsname~}%
        \fi
        \advance\@tempcnta\@ne
    \repeat}
```

As per a bug report by Benjamin Lings, we deactivate `\outer` definition of `^^L` temporarily (inside and outside of `\lst@ScanChars`) and restore the catcode at end of package via the `\lst@RestoreCatcodes` command.

```
\begingroup \catcode12=\active\let^^L\@empty
\gdef\lst@ScanChars{%
  \let\lsts@ssL^^L%
  \def^^L{\par}%
    \lst@GetChars\lst@RestoreOrigCatcodes\@ne {128}%
  \let^^L\lsts@ssL
    \lst@GetChars\lst@RestoreOrigExtendedCatcodes{128}{256}}
\endgroup
```

The scan can be issued by hand and at the beginning of a document.

```
\lst@Key{rescanchars}\relax{\lst@ScanChars}
\AtBeginDocument{\lst@ScanChars}
```

15.3.4 Adjusting the table

We begin with modifiers for the basic character classes.

`alsoletter`
`alsodigit`
`alsoother`

The macros `\lst@also`... will hold `\def`⟨*char*⟩`{`...`}` sequences, which adjusts the standard character table.

```
\lst@Key{alsoletter}\relax{%
    \lst@DoAlso{#1}\lst@alsoletter\lst@ProcessLetter}
\lst@Key{alsodigit}\relax{%
    \lst@DoAlso{#1}\lst@alsodigit\lst@ProcessDigit}
\lst@Key{alsoother}\relax{%
    \lst@DoAlso{#1}\lst@alsoother\lst@ProcessOther}
```

This is done at `SelectCharTable` and every language selection the macros get empty.

```
\lst@AddToHook{SelectCharTable}
    {\lst@alsoother \lst@alsodigit \lst@alsoletter}
\lst@AddToHookExe{SetLanguage}% init
    {\let\lst@alsoletter\@empty
     \let\lst@alsodigit\@empty
     \let\lst@alsoother\@empty}
```

The service macro starts a loop and ...

```
\def\lst@DoAlso#1#2#3{%
    \lst@DefOther\lst@arg{#1}\let#2\@empty
    \expandafter\lst@DoAlso@\expandafter#2\expandafter#3\lst@arg\relax}
\def\lst@DoAlso@#1#2#3{%
    \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`.

```
        \begingroup \lccode`\~=`#3\relax \lowercase{\endgroup
        \def\lst@temp##1\def~##2##3\relax{%
            \edef\lst@arg{\def\noexpand~{\noexpand#2\expandafter
                                         \noexpand\@gobble##2}}}}%
        \expandafter\lst@temp\lst@SelectStdCharTable\relax
        \lst@lExtend#1{\lst@arg}%
```

```
    \fi
    \lst@DoAlso@#1#2}
```

`\lst@SaveDef` **`\lst@DefSaveDef`** **`\lst@LetSaveDef`** These macros can be used in language definitions to make special changes. They save the definition and define or assign a new one.

```
\def\lst@SaveDef#1#2{%
    \begingroup \lccode`\~=#1\relax \lowercase{\endgroup\let#2~}}
\def\lst@DefSaveDef#1#2{%
    \begingroup \lccode`\~=#1\relax \lowercase{\endgroup\let#2~\def~}}
\def\lst@LetSaveDef#1#2{%
    \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 ⟨*1st*⟩`{`⟨*2nd*⟩`}{`⟨*rest*⟩`}` and say that this input string is 'replaced' by ⟨*save 1st*⟩`{`⟨*2nd*⟩`}{`⟨*rest*⟩`}`—plus ⟨*execute*⟩, ⟨*pre*⟩, and ⟨*post*⟩. This main work is done by `\lst@CDefIt`.

```
\def\lst@CDef#1{\lst@CDef@#1}
\def\lst@CDef@#1#2#3#4{\lst@CDefIt#1{#2}{#3}{#4#2#3}#4}
```

`\lst@CDefX` drops the input string.

```
\def\lst@CDefX#1{\lst@CDefX@#1}
\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!

```
\def\lst@CDefIt#1#2#3#4#5#6#7#8{%
    \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.

```
        \def#1{#6\def\lst@next{#7#4#8}\lst@next}%
    \else \ifx\@empty#3\@empty
```

For a two character sequence we test whether ⟨*pre*⟩ and ⟨*post*⟩ must be executed.

```
        \def#1##1{%
            #6%
            \ifx##1#2\def\lst@next{#7#4#8}\else
                     \def\lst@next{#5##1}\fi
            \lst@next}%
    \else
```

We do the same for an arbitrary character sequence—except that we have to use `\lst@IfNextCharsArg` instead of `\ifx...\fi`.

```
        \def#1{%
            #6%
            \lst@IfNextCharsArg{#2#3}{#7#4#8}%
                                     {\expandafter#5\lst@eaten}}%
    \fi \fi}
```

`\lst@CArgX` We make `#1#2` active and call `\lst@CArg`.

```
\def\lst@CArgX#1#2\relax{%
    \lst@DefActive\lst@arg{#1#2}%
    \expandafter\lst@CArg\lst@arg\relax}
```

`\lst@CArg` arranges the first two arguments for `\lst@CDef`[`X`]. We get an undefined macro and use `\@empty\@empty\relax` as delimiter for the submacro.

```
\def\lst@CArg#1#2\relax{%
    \lccode`\/=`#1\lowercase{\def\lst@temp{/}}%
    \lst@GetFreeMacro{lst@c\lst@temp}%
    \expandafter\lst@CArg@\lst@freemacro#1#2\@empty\@empty\relax}
```

Save meaning of ⟨*1st*⟩=`#2` in ⟨*save 1st*⟩=`#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`.

```
\def\lst@CArg@#1#2#3#4\@empty#5\relax#6{%
    \let#1#2%
    \ifx\@empty#3\@empty
        \def\lst@next{#6{#2{}{}}}%
    \else
        \def\lst@next{#6{#2#3{#4}}}%
    \fi
    \lst@next #1}
```

`\lst@CArgEmpty` ‘executes’ an `\@empty`-delimited argument. We will use it for the delimiters.

```
\def\lst@CArgEmpty#1\@empty{#1}
```

15.4 Delimiters

Here we start with general definitions common to all delimiters.

`excludedelims` controls which delimiters are not printed in ⟨*whatever*⟩style. We just define `\lst@ifex`⟨*whatever*⟩ to be true. Such switches are set false in the `ExcludeDelims` hook and are handled by the individual delimiters.

```
\lst@Key{excludedelims}\relax
    {\lsthk@ExcludeDelims \lst@NormedDef\lst@temp{#1}%
     \expandafter\lst@for\lst@temp\do
     {\expandafter\let\csname\@lst @ifex##1\endcsname\iftrue}}
```

`\lst@DelimPrint` And this macro might help in doing so. `#1` is `\lst@ifex`⟨*whatever*⟩ (plus `\else`) or just `\iffalse`, and `#2` will be the delimiter. The temporary mode change ensures that the characters can't end the current delimiter or start a new one.

```
\def\lst@DelimPrint#1#2{%
    #1%
      \begingroup
        \lst@mode\lst@nomode \lst@modetrue
        #2\lst@XPrintToken
      \endgroup
      \lst@ResetToken
    \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.

```
\def\lst@DelimOpen#1#2#3#4#5#6\@empty{%
    \lst@TrackNewLines \lst@XPrintToken
    \lst@DelimPrint#1{#6}%
    \lst@EnterMode{#4}{\def\lst@currstyle#5}%
    \lst@DelimPrint{#1#2}{#6}%
    #3}
```

`\lst@DelimClose` is the same in reverse order.

```
\def\lst@DelimClose#1#2#3\@empty{%
    \lst@TrackNewLines \lst@XPrintToken
    \lst@DelimPrint{#1#2}{#3}%
    \lst@LeaveMode
    \lst@DelimPrint{#1}{#3}}
```

`\lst@BeginDelim` `\lst@EndDelim` These definitions are applications of `\lst@DelimOpen` and `\lst@DelimClose`: the delimiters have the same style as the delimited text.

```
\def\lst@BeginDelim{\lst@DelimOpen\iffalse\else{}}
\def\lst@EndDelim{\lst@DelimClose\iffalse\else}
```

`\lst@BeginIDelim` `\lst@EndIDelim` Another application: no delimiter is printed.

```
\def\lst@BeginIDelim{\lst@DelimOpen\iffalse{}{}}
\def\lst@EndIDelim{\lst@DelimClose\iffalse{}}
```

`\lst@DefDelims` This macro defines all delimiters and is therefore reset every language selection.

```
\lst@AddToHook{SelectCharTable}{\lst@DefDelims}
\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*⟩.

```
\def\lst@Delim#1{%
    \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`.

```
    \@ifstar{\@ifstar{\lst@Delim@{#1}}%
                     {\let\lst@cumulative\relax
                      \lst@Delim@{#1}}}%
            {\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`.

```
\def\lst@Delim@#1[#2]{%
    \gdef\lst@delimtype{#2}%
    \@ifnextchar[\lst@Delim@sty
                 {\lst@Delim@sty[#1]}}
\def\lst@Delim@sty[#1]{%
    \def\lst@delimstyle{#1}%
    \ifx\@empty#1\@empty\else
        \lst@Delim@sty@ #1\@nil
    \fi
    \@ifnextchar[\lst@Delim@option
                 \lst@Delim@delim}
\def\lst@Delim@option[#1]{\def\lst@arg{[#1]}\lst@Delim@delim}
```

`[` and `]` in the replacement text above have been added after a bug report by Stephen Reindl.

The definition of `\lst@delimstyle` depends on whether the first token is a control sequence. Here we possibly build `\lst@`⟨*style*⟩.

```
\def\lst@Delim@sty@#1#2\@nil{%
    \if\relax\noexpand#1\else
        \edef\lst@delimstyle{\expandafter\noexpand
                            \csname\@lst @\lst@delimstyle\endcsname}%
    \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.

```
\def\lst@Delim@delim#1\relax#2#3#4#5#6#7#8{%
    \ifx #4\@empty \lst@Delim@delall{#2}\fi
    \ifx\@empty#1\@empty
        \ifx #4\@nil
            \@ifundefined{\@lst @#2DM@\lst@delimtype}%
                {\lst@Delim@delall{#2@\lst@delimtype}}%
                {\lst@Delim@delall{#2DM@\lst@delimtype}}%
        \fi
    \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.

```
        \expandafter\lst@Delim@args\expandafter
             {\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`.

```
        \let\lst@delim\@empty
        \expandafter\lst@IfOneOf\lst@delimtype\relax#3%
        {\@ifundefined{\@lst @#2DM@\lst@delimtype}%
             {\lst@lExtend\lst@delim{\csname\@lst @#2@\lst@delimtype
                                    \expandafter\endcsname\lst@arg}}%
             {\lst@lExtend\lst@delim{\expandafter\lst@UseDynamicMode
                                    \csname\@lst @#2DM@\lst@delimtype
                                    \expandafter\endcsname\lst@arg}}%
```

Now, depending on the mode `#4` we either remove this particular delimiter or append it to all current ones.

```
         \ifx #4\@nil
             \let\lst@temp\lst@DefDelims \let\lst@DefDelims\@empty
             \expandafter\lst@Delim@del\lst@temp\@empty\@nil\@nil\@nil
         \else
             \lst@lExtend\lst@DefDelims\lst@delim
         \fi}%
```

An unknown type issues an error.

```
        {\PackageError{Listings}{Illegal type `\lst@delimtype'}%
                               {#2 types are #3.}}%
     \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`, ...

```
\def\lst@Delim@args#1#2#3#4#5#6#7{%
    \begingroup
    \lst@false \let\lst@next\lst@XConvert
```

... we remove that character from `\lst@delimtype`, and `#5` might select a different conversion setting or macro.

```
    \@ifnextchar #4{\xdef\lst@delimtype{\expandafter\@gobble
                                        \lst@delimtype}%
                    #5\lst@next#2\@nil
                    \lst@lAddTo\lst@arg{\@empty#6}%
                    \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.

```
                   {\lst@next#2\@nil
                    \lst@lAddTo\lst@arg{\@empty#3}%
                    \lst@GobbleNil}%
                 #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.

```
    \global\let\@gtempa\lst@arg
    \endgroup
    \let\lst@arg\@gtempa
    \ifx #7\@nil\else
        \expandafter\lst@Delim@args@\expandafter{\lst@delimstyle}%
    \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*⟩.

```
\def\lst@Delim@args@#1{%
    \lst@if
        \lst@lAddTo\lst@arg{{{#1}\lst@modetrue}}%
    \else
        \ifx\lst@cumulative\@empty
            \lst@lAddTo\lst@arg{{{}#1}}%
        \else
            \lst@lAddTo\lst@arg{{{#1}}}%
        \fi
    \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.

```
\def\lst@Delim@del#1\@empty#2#3#4{%
    \ifx #2\@nil\else
        \def\lst@temp{#1\@empty#2#3}%
        \ifx\lst@temp\lst@delim\else
```

```
            \lst@lAddTo\lst@DefDelims{#1\@empty#2#3{#4}}%
        \fi
        \expandafter\lst@Delim@del
    \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.

```
\def\lst@Delim@delall#1{%
    \begingroup
    \edef\lst@delim{\expandafter\string\csname\@lst @#1\endcsname}%
    \lst@false \global\let\@gtempa\@empty
    \expandafter\lst@Delim@delall@\lst@DefDelims\@empty
    \endgroup
    \let\lst@DefDelims\@gtempa}
```

We first discard a preceding `\lst@UseDynamicMode`.

```
\def\lst@Delim@delall@#1{%
    \ifx #1\@empty\else
        \ifx #1\lst@UseDynamicMode
            \lst@true
            \let\lst@next\lst@Delim@delall@do
        \else
            \def\lst@next{\lst@Delim@delall@do#1}%
        \fi
        \expandafter\lst@next
    \fi}
```

Then we can check whether (the following) `\lst@`⟨*delimiter name*⟩… matches the delimiter class given by `\lst@delim`.

```
\def\lst@Delim@delall@do#1#2\@empty#3#4#5{%
    \expandafter\lst@IfSubstring\expandafter{\lst@delim}{\string#1}%
      {}%
      {\lst@if \lst@AddTo\@gtempa\lst@UseDynamicMode \fi
       \lst@AddTo\@gtempa{#1#2\@empty#3#4{#5}}}%
    \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`.

```
\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}
```

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.

```
\gdef\lst@DefDelimE#1#2#3#4#5#6#7{%
    \lst@CDef{#1}#2%
```

```
        {#3}%
        {\let\lst@enext\lst@CArgEmpty
         \ifnum #7=\lst@mode%
             #4%
             \let\lst@enext#6%
         \else
             #5%
         \fi
         \lst@enext}%
        \@empty}
```

```
\lst@AddToHook{Init}{\let\lst@bnext\relax \let\lst@enext\relax}
```

`\lst@DefDelimBE` This service macro will actually define all string delimiters.

```
\gdef\lst@DefDelimBE#1#2#3#4#5#6#7#8#9{%
    \lst@CDef{#1}#2%
        {#3}%
        {\let\lst@bnext\lst@CArgEmpty
         \ifnum #7=\lst@mode
             #4%
             \let\lst@bnext#9%
         \else
             \lst@ifmode\else
                 #5%
                 \def\lst@bnext{#6{#7}{#8}}%
             \fi
         \fi
         \lst@bnext}%
        \@empty}
```

`\lst@delimtypes` is the list of general delimiter types.

```
\gdef\lst@delimtypes{s,l}
```

`\lst@DelimKey` We just put together the arguments for `\lst@Delim`.

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

`delim`
`moredelim`
`deletedelim` all use `\lst@DelimKey`.

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

`\lst@DelimDM@l`
`\lst@DelimDM@s` Nohting special here.

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

```
\gdef\lst@DelimDM@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}}
```

```
⟨/kernel⟩
```

15.4.1 Strings

Just starting a new aspect.

```
⟨*misc⟩
\lst@BeginAspect{strings}
```

`\lst@stringtypes` is the list of . . . string types? The several valid string types are defined on page 49.

```
\gdef\lst@stringtypes{d,b,m,bd,db,s}
```

`\lst@StringKey` We just put together the arguments for `\lst@Delim`.

```
\gdef\lst@StringKey#1#2{%
    \lst@Delim\lst@stringstyle #2\relax
        {String}\lst@stringtypes #1%
                     {\lst@BeginString\lst@EndString}%
        \@@end\@empty{}}
```

`string`
`morestring`
`deletestring` all use `\lst@StringKey`.

```
\lst@Key{string}\relax{\lst@StringKey\@empty{#1}}
\lst@Key{morestring}\relax{\lst@StringKey\relax{#1}}
\lst@Key{deletestring}\relax{\lst@StringKey\@nil{#1}}
```

`stringstyle` You shouldn't need comments on the following two lines, do you?

```
\lst@Key{stringstyle}{}{\def\lst@stringstyle{#1}}
\lst@AddToHook{EmptyStyle}{\let\lst@stringstyle\@empty}
```

`showstringspaces` Thanks to Knut Müller for reporting problems with `\blankstringtrue` (now `showstringspaces=false`). The problem has gone.

```
\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`.

```
\gdef\lst@BeginString{%
    \lst@DelimOpen
        \lst@ifexstrings\else
        {\lst@ifshowstringspaces
             \lst@keepspacestrue
             \let\lst@outputspace\lst@visiblespace
         \fi}}
```

```
\lst@AddToHookExe{ExcludeDelims}{\let\lst@ifexstrings\iffalse}
```

`\lst@EndString` Again the two tokens following `\lst@DelimClose` are arguments.

```
\gdef\lst@EndString{\lst@DelimClose\lst@ifexstrings\else}
```

And now all the `\lst@StringDM@`⟨*type*⟩ definitions.

`\lst@StringDM@d` 'd' means no extra work.; the first three arguments after `\lst@DefDelimBE` are left empty. The others are used to start and end the string.

```
\gdef\lst@StringDM@d#1#2\@empty#3#4#5{%
    \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.

```
\gdef\lst@StringDM@b#1#2\@empty#3#4#5{%
    \let\lst@ifbstring\iftrue
    \lst@CArg #2\relax\lst@DefDelimBE
       {\lst@ifletter \lst@Output \lst@letterfalse \fi}%
       {\ifx\lst@lastother\lstum@backslash
            \expandafter\@gobblethree
        \fi}{}#3{#1}{#5}#4}
```

```
\global\let\lst@ifbstring\iffalse % init
```

Heiko Heil reported problems with double backslashes. So:

```
\lst@AddToHook{SelectCharTable}{%
    \lst@ifbstring
        \lst@CArgX \\\\\relax \lst@CDefX{}%
           {\lst@ProcessOther\lstum@backslash
            \lst@ProcessOther\lstum@backslash
            \let\lst@lastother\relax}%
           {}%
    \fi}
```

The reset of `\lst@lastother` has been added after a bug reports by Hermann Hüttler and Dan Luecking.

`\lst@StringDM@bd` `\lst@StringDM@db` are just the same and the same as `\lst@StringDM@b`.

```
\global\let\lst@StringDM@bd\lst@StringDM@b
\global\let\lst@StringDM@db\lst@StringDM@bd
```

`\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.

```
\gdef\lst@StringDM@m#1#2\@empty#3#4#5{%
    \lst@CArg #2\relax\lst@DefDelimBE{}{}%
        {\let\lst@next\@gobblethree
         \lst@ifletter\else
             \lst@IfLastOtherOneOf{)].0123456789\lstum@rbrace'}%
                 {}%
                 {\let\lst@next\@empty}%
         \fi
         \lst@next}#3{#1}{#5}#4}
```

`\lst@StringDM@s` is for string-delimited strings, just as for comments. This is needed for Ruby, and possibly other languages.

```
\gdef\lst@StringDM@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}}
```

`\lstum@rbrace` This has been used above.

```
\lst@SaveOutputDef{"7D}\lstum@rbrace
```

```
\lst@EndAspect
⟨/misc⟩
```

For MetaFont and MetaPost we now define macros to print the input-filenames in stringstyle.

```
⟨*misc⟩
\lst@BeginAspect{mf}
```

`\lst@mfinputmode`
`\lst@String@mf`

```
\lst@AddTo\lst@stringtypes{,mf}
\lst@NewMode\lst@mfinputmode
```

```
\gdef\lst@String@mf#1\@empty#2#3#4{%
  \lst@CArg #1\relax\lst@DefDelimB
       {}{}{\lst@ifletter \expandafter\@gobblethree \fi}%
       \lst@BeginStringMFinput\lst@mfinputmode{#4\lst@Lmodetrue}%
  \@ifundefined{lsts@semicolon}%
  {\lst@DefSaveDef{`\;}\lsts@semicolon{% ; and space end the filename
      \ifnum\lst@mode=\lst@mfinputmode
          \lst@XPrintToken
          \expandafter\lst@LeaveMode
      \fi
      \lsts@semicolon}%
   \lst@DefSaveDef{`\ }\lsts@space{%
      \ifnum\lst@mode=\lst@mfinputmode
          \lst@XPrintToken
          \expandafter\lst@LeaveMode
      \fi
      \lsts@space}%
  }{}}
```

`\lst@BeginStringMFinput` It remains to define this macro. In contrast to `\lst@PrintDelim`, we don't use `\lst@modetrue` to allow keyword detection here.

```
\gdef\lst@BeginStringMFinput#1#2#3\@empty{%
    \lst@TrackNewLines \lst@XPrintToken
      \begingroup
        \lst@mode\lst@nomode
        #3\lst@XPrintToken
      \endgroup
      \lst@ResetToken
    \lst@EnterMode{#1}{\def\lst@currstyle#2}%
    \lst@ifshowstringspaces
         \lst@keepspacestrue
         \let\lst@outputspace\lst@visiblespace
    \fi}
```

```
\lst@EndAspect
⟨/misc⟩
```

15.4.2 Comments

That's what we are working on.

```
⟨*misc⟩
\lst@BeginAspect{comments}
```

`\lst@commentmode` is a general purpose mode for comments.

```
\lst@NewMode\lst@commentmode
```

`\lst@commenttypes` Via `comment` available comment types: **l**ine, **f**ixed column, **s**ingle, and **n**ested and all with preceding **i** for invisible comments.

```
\gdef\lst@commenttypes{l,f,s,n}
```

`\lst@CommentKey` We just put together the arguments for `\lst@Delim`.

```
\gdef\lst@CommentKey#1#2{%
    \lst@Delim\lst@commentstyle #2\relax
        {Comment}\lst@commenttypes #1%
                {\lst@BeginComment\lst@EndComment}%
        i\@empty{\lst@BeginInvisible\lst@EndInvisible}}
```

`comment`
`morecomment`
`deletecomment`

The keys are easy since defined in terms of `\lst@CommentKey`.

```
\lst@Key{comment}\relax{\lst@CommentKey\@empty{#1}}
\lst@Key{morecomment}\relax{\lst@CommentKey\relax{#1}}
\lst@Key{deletecomment}\relax{\lst@CommentKey\@nil{#1}}
```

`commentstyle` Any hints necessary?

```
\lst@Key{commentstyle}{}{\def\lst@commentstyle{#1}}
\lst@AddToHook{EmptyStyle}{\let\lst@commentstyle\itshape}
```

`\lst@BeginComment`
`\lst@EndComment`

Once more the three tokens following `\lst@DelimOpen` are arguments.

```
\gdef\lst@BeginComment{%
    \lst@DelimOpen
        \lst@ifexcomments\else
        \lsthk@AfterBeginComment}
```

Ditto.

```
\gdef\lst@EndComment{\lst@DelimClose\lst@ifexcomments\else}
```

```
\lst@AddToHook{AfterBeginComment}{}
\lst@AddToHookExe{ExcludeDelims}{\let\lst@ifexcomments\iffalse}
```

`\lst@BeginInvisible`
`\lst@EndInvisible`

Print preceding characters and begin dropping the output.

```
\gdef\lst@BeginInvisible#1#2#3\@empty{%
    \lst@TrackNewLines \lst@XPrintToken
    \lst@BeginDropOutput{#1}}
```

Don't print the delimiter and end dropping the output.

```
\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.

```
\gdef\lst@CommentDM@l#1#2\@empty#3#4#5{%
    \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.

```
\gdef\lst@CommentDM@f#1{%
    \@ifnextchar[{\lst@Comment@@f{#1}}%
                {\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).

```
\gdef\lst@Comment@@f#1[#2]#3\@empty#4#5#6{%
    \lst@CArg #3\relax\lst@DefDelimB{}{}%
        {\lst@CalcColumn
         \ifnum #2=\@tempcnta\else
             \expandafter\@gobblethree
         \fi}%
        #4{#1}{#6\lst@Lmodetrue}}
```

`\lst@CommentDM@s` Nothing special here.

```
\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}}
```

`\lst@CommentDM@n` We either give an error message or define the nested comment.

```
\gdef\lst@CommentDM@n#1#2#3\@empty#4#5#6{%
    \ifx\@empty#3\@empty\else
        \def\@tempa{#2}\def\@tempb{#3}%
        \ifx\@tempa\@tempb
            \PackageError{Listings}{Identical delimiters}%
            {These delimiters make no sense with nested comments.}%
        \else
            \lst@CArg #2\relax\lst@DefDelimB
                {}%
```

Note that the following `\@gobble` eats an `\else` from `\lst@DefDelimB`.

```
                {\ifnum\lst@mode=#1\relax \expandafter\@gobble \fi}%
                {}#4{#1}{#6}%
            \lst@CArg #3\relax\lst@DefDelimE{}{}{}#5{#1}%
        \fi
    \fi}
```

```
\lst@EndAspect
⟨/misc⟩
```

15.4.3 PODs

PODs are defined as a separate aspect.

```
⟨*misc⟩
\lst@BeginAspect{pod}
```

`printpod` `podcomment` We begin with the user keys, which I introduced after communication with Michael Piotrowski.

```
\lst@Key{printpod}{false}[t]{\lstKV@SetIf{#1}\lst@ifprintpod}
\lst@Key{podcomment}{false}[t]{\lstKV@SetIf{#1}\lst@ifpodcomment}
\lst@AddToHookExe{SetLanguage}{\let\lst@ifpodcomment\iffalse}
```

`\lst@PODmode` is the static mode for PODs.

```
\lst@NewMode\lst@PODmode
```

We adjust some characters if the user has selected `podcomment=true`.

```
\lst@AddToHook{SelectCharTable}
    {\lst@ifpodcomment
         \lst@CArgX =\relax\lst@DefDelimB{}{}%
```

The following code is executed if we've found an equality sign and haven't entered a mode (in fact if mode changes are allowed): We 'begin drop output' and gobble the usual begin of comment sequence (via `\@gobblethree`) if PODs aren't be printed. Moreover we gobble it if the current column number is not zero—`\@tempcnta` is valued below.

```
           {\ifnum\@tempcnta=\z@
                \lst@ifprintpod\else
                    \def\lst@bnext{\lst@BeginDropOutput\lst@PODmode}%
                    \expandafter\expandafter\expandafter\@gobblethree
                \fi
            \else
               \expandafter\@gobblethree
            \fi}%
           \lst@BeginComment\lst@PODmode{{\lst@commentstyle}}%
```

If we come to `=`, we calculate the current column number (zero based).

```
         \lst@CArgX =cut\^^M\relax\lst@DefDelimE
           {\lst@CalcColumn}%
```

If there is additionally `cut`+EOL and if we are in `\lst@PODmode` but not in column one, we must gobble the 'end comment sequence'.

```
           {\ifnum\@tempcnta=\z@\else
                \expandafter\@gobblethree
            \fi}%
           {}%
           \lst@EndComment\lst@PODmode
     \fi}
```

```
\lst@EndAspect
⟨/misc⟩
```

15.4.4 Tags

Support for HTML and other 'markup languages'.

```
⟨*misc⟩
\lst@BeginAspect[keywords]{html}
```

`\lst@tagtypes` Again we begin with the list of tag types. It's rather short.

```
\gdef\lst@tagtypes{s}
```

`\lst@TagKey` Again we just put together the arguments for `\lst@Delim` and ...

```
\gdef\lst@TagKey#1#2{%
    \lst@Delim\lst@tagstyle #2\relax
        {Tag}\lst@tagtypes #1%
                     {\lst@BeginTag\lst@EndTag}%
        \@@end\@empty{}}
```

`tag` ... we use the definition here.

```
\lst@Key{tag}\relax{\lst@TagKey\@empty{#1}}
```

`tagstyle` You shouldn't need comments on the following two lines, do you?

```
\lst@Key{tagstyle}{}{\def\lst@tagstyle{#1}}
\lst@AddToHook{EmptyStyle}{\let\lst@tagstyle\@empty}
```

`\lst@BeginTag` The special things here are: (1) We activate keyword detection inside tags and (2) we initialize the switch `\lst@iffirstintag` if necessary.

```
\gdef\lst@BeginTag{%
    \lst@DelimOpen
        \lst@ifextags\else
        {\let\lst@ifkeywords\iftrue
         \lst@ifmarkfirstintag \lst@firstintagtrue \fi}}
```

```
\lst@AddToHookExe{ExcludeDelims}{\let\lst@ifextags\iffalse}
```

`\lst@EndTag` is just like the other `\lst@End`⟨*whatever*⟩ definitions.

```
\gdef\lst@EndTag{\lst@DelimClose\lst@ifextags\else}
```

`usekeywordsintag` `markfirstintag` The second key has already been 'used'.

```
\lst@Key{usekeywordsintag}t[t]{\lstKV@SetIf{#1}\lst@ifusekeysintag}
\lst@Key{markfirstintag}f[t]{\lstKV@SetIf{#1}\lst@ifmarkfirstintag}
```

For this, we install a (global) switch, ...

```
\gdef\lst@firstintagtrue{\global\let\lst@iffirstintag\iftrue}
\global\let\lst@iffirstintag\iffalse
```

... which is reset by the output of an identifier but not by other output.

```
\lst@AddToHook{PostOutput}{\lst@tagresetfirst}
\lst@AddToHook{Output}
    {\gdef\lst@tagresetfirst{\global\let\lst@iffirstintag\iffalse}}
\lst@AddToHook{OutputOther}{\gdef\lst@tagresetfirst{}}
```

Now we only need to test against this switch in the `Output` hook.

```
\lst@AddToHook{Output}
    {\ifnum\lst@mode=\lst@tagmode
          \lst@iffirstintag \let\lst@thestyle\lst@gkeywords@sty \fi
```

Moreover we check here, whether the keyword style is always to be used.

```
          \lst@ifusekeysintag\else \let\lst@thestyle\lst@gkeywords@sty\fi
     \fi}
```

`\lst@tagmode` We allocate the mode and ...

```
\lst@NewMode\lst@tagmode
```

deactivate keyword detection if any tag delimiter is defined (see below).

```
\lst@AddToHook{Init}{\global\let\lst@ifnotag\iftrue}
\lst@AddToHook{SelectCharTable}{\let\lst@ifkeywords\lst@ifnotag}
```

`\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.

```
\gdef\lst@Tag@s#1#2\@empty#3#4#5{%
    \global\let\lst@ifnotag\iffalse
    \lst@CArg #1\relax\lst@DefDelimB {}{}%
        {\ifnum\lst@mode=\lst@tagmode \expandafter\@gobblethree \fi}%
        #3\lst@tagmode{#5}%
    \lst@CArg #2\relax\lst@DefDelimE {}{}{}#4\lst@tagmode}%
```

`\lst@BeginCDATA` This macro is used by the XML language definition.

```
\gdef\lst@BeginCDATA#1\@empty{%
    \lst@TrackNewLines \lst@PrintToken
    \lst@EnterMode\lst@GPmode{}\let\lst@ifmode\iffalse
    \lst@mode\lst@tagmode #1\lst@mode\lst@GPmode\relax\lst@modetrue}
```

```
\lst@EndAspect
⟨/misc⟩
```

15.5 Replacing input

```
⟨*kernel⟩
```

`\lst@ReplaceInput` is defined in terms of `\lst@CArgX` and `\lst@CDefX`.

```
\def\lst@ReplaceInput#1{\lst@CArgX #1\relax\lst@CDefX{}{}}
```

`literate` Jason Alexander asked for something like that. The key looks for a star and saves the argument.

```
\def\lst@Literatekey#1\@nil@{\let\lst@ifxliterate\lst@if
                               \def\lst@literate{#1}}
\lst@Key{literate}{}{\@ifstar{\lst@true \lst@Literatekey}
                               {\lst@false\lst@Literatekey}#1\@nil@}
\lst@AddToHook{SelectCharTable}
    {\ifx\lst@literate\@empty\else
         \expandafter\lst@Literate\lst@literate{}\relax\z@
     \fi}
```

Internally we don't make use of the 'replace input' feature any more.

```
\def\lst@Literate#1#2#3{%
    \ifx\relax#2\@empty\else
        \lst@CArgX #1\relax\lst@CDef
            {}
            {\let\lst@next\@empty
             \lst@ifxliterate
                 \lst@ifmode \let\lst@next\lst@CArgEmpty \fi
             \fi
             \ifx\lst@next\@empty
                 \ifx\lst@OutputBox\@gobble\else
                   \lst@XPrintToken \let\lst@scanmode\lst@scan@m
                   \lst@token{#2}\lst@length#3\relax
                   \lst@XPrintToken
                 \fi
                 \let\lst@next\lst@CArgEmptyGobble
             \fi
             \lst@next}%
            \@empty
        \expandafter\lst@Literate
    \fi}
\def\lst@CArgEmptyGobble#1\@empty{}
```

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.

```
\def\lst@BeginDropInput#1{%
    \lst@EnterMode{#1}%
    {\lst@modetrue
     \let\lst@OutputBox\@gobble
     \let\lst@ifdropinput\iftrue
     \let\lst@ProcessLetter\@gobble
     \let\lst@ProcessDigit\@gobble
     \let\lst@ProcessOther\@gobble
     \let\lst@ProcessSpace\@empty
     \let\lst@ProcessTabulator\@empty
     \let\lst@ProcessFormFeed\@empty}}
\let\lst@ifdropinput\iffalse % init
```

```
⟨/kernel⟩
```

15.6 Escaping to LaTeX

We now define the ... damned ... the aspect has escaped!

```
⟨*misc⟩
\lst@BeginAspect{escape}
```

`texcl` Communication with Jörn Wilms is responsible for this key. The definition and the first hooks are easy.

```
\lst@Key{texcl}{false}[t]{\lstKV@SetIf{#1}\lst@iftexcl}
\lst@AddToHook{TextStyle}{\let\lst@iftexcl\iffalse}
\lst@AddToHook{EOL}
    {\ifnum\lst@mode=\lst@TeXLmode
         \expandafter\lst@escapeend
         \expandafter\lst@LeaveAllModes
         \expandafter\lst@ReenterModes
     \fi}
```

If the user wants TeX comment lines, we print the comment separator and interrupt the normal processing.

```
\lst@AddToHook{AfterBeginComment}
    {\lst@iftexcl \lst@ifLmode \lst@ifdropinput\else
         \lst@PrintToken
         \lst@LeaveMode \lst@InterruptModes
         \lst@EnterMode{\lst@TeXLmode}{\lst@modetrue\lst@commentstyle}%
         \expandafter\expandafter\expandafter\lst@escapebegin
     \fi \fi \fi}
```

```
\lst@NewMode\lst@TeXLmode
```

`\lst@ActiveCDefX` Same as `\lst@CDefX` but we both make `#1` active and assign a new catcode.

```
\gdef\lst@ActiveCDefX#1{\lst@ActiveCDefX@#1}
\gdef\lst@ActiveCDefX@#1#2#3{
    \catcode`#1\active\lccode`\~=`#1%
    \lowercase{\lst@CDefIt~}{#2}{#3}{}}
```

`\lst@Escape` gets four arguments all in all. The first and second are the ‘begin’ and ‘end’ escape sequences, the third is executed when the escape starts, and the fourth right before ending it. We use the same mechanism as for TeX comment lines.

The `\lst@ifdropinput` test has been added after a bug report by Michael Weber. The `\lst@newlines\z@` was added due to a bug report by Frank Atanassow.

```
\gdef\lst@Escape#1#2#3#4{%
    \lst@CArgX #1\relax\lst@CDefX
        {}%
        {\lst@ifdropinput\else
         \lst@TrackNewLines\lst@OutputLostSpace \lst@XPrintToken
         \lst@InterruptModes
         \lst@EnterMode{\lst@TeXmode}{\lst@modetrue}%
```

Now we must define the character sequence to end the escape.

```
         \ifx\^^M#2%
             \lst@CArg #2\relax\lst@ActiveCDefX
                 {}%
                 {\lst@escapeend #4\lst@LeaveAllModes\lst@ReenterModes}%
                 {\lst@MProcessListing}%
         \else
             \lst@CArg #2\relax\lst@ActiveCDefX
                 {}%
                 {\lst@escapeend #4\lst@LeaveAllModes\lst@ReenterModes
                  \lst@newlines\z@ \lst@whitespacefalse}%
                 {}%
         \fi
         #3\lst@escapebegin
         \fi}%
        {}}
```

The `\lst@whitespacefalse` above was added after a bug report from Martin Steffen.

```
\lst@NewMode\lst@TeXmode
```

`escapebegin`
`escapeend`

The keys simply store the arguments.

```
\lst@Key{escapebegin}{}{\def\lst@escapebegin{#1}}
\lst@Key{escapeend}{}{\def\lst@escapeend{#1}}
```

`escapechar`

The introduction of this key is due to a communication with Rui Oliveira. We define `\lst@DefEsc` and execute it after selecting the standard character table.

```
\lst@Key{escapechar}{}
    {\ifx\@empty#1\@empty
         \let\lst@DefEsc\relax
     \else
         \def\lst@DefEsc{\lst@Escape{#1}{#1}{}{}}%
    \fi}
\lst@AddToHook{TextStyle}{\let\lst@DefEsc\@empty}
\lst@AddToHook{SelectCharTable}{\lst@DefEsc}
```

`escapeinside`

Nearly the same.

```
\lst@Key{escapeinside}{}{\lstKV@TwoArg{#1}%
    {\let\lst@DefEsc\@empty
     \ifx\@empty##1@empty\else \ifx\@empty##2\@empty\else
         \def\lst@DefEsc{\lst@Escape{##1}{##2}{}{}}%
     \fi\fi}}
```

`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.

```
\lst@Key{mathescape}{false}[t]{\lstKV@SetIf{#1}\lst@ifmathescape}
\lst@AddToHook{SelectCharTable}
    {\lst@ifmathescape \lst@Escape{\$}{\$}%
        {\setbox\@tempboxa=\hbox\bgroup$}%
        {$\egroup \lst@CalcLostSpaceAndOutput}\fi}
```

```
\lst@EndAspect
⟨/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.

```
⟨*misc⟩
\lst@BeginAspect{keywords}
```

```
\global\let\lst@ifsensitive\iftrue % init
\global\let\lst@ifsensitivedefed\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.

```
\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`.

```
\gdef\lst@KeywordTest#1#2#3{%
    \begingroup \let\lst@UM\@empty
    \global\expandafter\let\expandafter\@gtempa
        \csname\@lst#1@\the\lst@token\endcsname
    \endgroup
    \ifx\@gtempa\relax\else
        \let\lst@thestyle\@gtempa
    \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`.

```
\gdef\lst@KEYWORDTEST{%
    \uppercase\expandafter{\expandafter
        \lst@KEYWORDTEST@\the\lst@token}\relax}
\gdef\lst@KEYWORDTEST@#1\relax#2#3#4{%
```

```
    \begingroup \let\lst@UM\@empty
    \global\expandafter\let\expandafter\@gtempa
        \csname\@lst#2@#1\endcsname
    \endgroup
    \ifx\@gtempa\relax\else
        \let\lst@thestyle\@gtempa
    \fi}
```

`\lst@WorkingTest`
`\lst@WORKINGTEST`
The same except that `\lst`⟨*prefix*⟩`@`⟨*current token*⟩ might be a working procedure; it is executed.

```
\gdef\lst@WorkingTest#1#2#3{%
    \begingroup \let\lst@UM\@empty
    \global\expandafter\let\expandafter\@gtempa
        \csname\@lst#1@\the\lst@token\endcsname
    \endgroup
    \@gtempa}
```

```
\gdef\lst@WORKINGTEST{%
    \uppercase\expandafter{\expandafter
        \lst@WORKINGTEST@\the\lst@token}\relax}
\gdef\lst@WORKINGTEST@#1\relax#2#3#4{%
    \begingroup \let\lst@UM\@empty
    \global\expandafter\let\expandafter\@gtempa
        \csname\@lst#2@#1\endcsname
    \endgroup
    \@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, . . .

```
\gdef\lst@DefineKeywords#1#2#3{%
    \lst@ifsensitive
        \def\lst@next{\lst@for#2}%
    \else
        \def\lst@next{\uppercase\expandafter{\expandafter\lst@for#2}}%
    \fi
    \lst@next\do
```

. . . iterate through the list, and make `\lst`⟨*prefix*⟩`@`⟨*keyword*⟩ (if undefined) equivalent to `\lst@g`⟨*name*⟩`@sty` which is possibly a working macro.

```
    {\expandafter\ifx\csname\@lst#1@##1\endcsname\relax
        \global\expandafter\let\csname\@lst#1@##1\endcsname#3%
     \fi}}
```

`\lst@UndefineKeywords`
We make the keywords upper case if necessary, . . .

```
\gdef\lst@UndefineKeywords#1#2#3{%
    \lst@ifsensitivedefed
        \def\lst@next{\lst@for#2}%
    \else
        \def\lst@next{\uppercase\expandafter{\expandafter\lst@for#2}}%
    \fi
    \lst@next\do
```

. . . iterate through the list, and 'undefine' `\lst`⟨*prefix*⟩`@`⟨*keyword*⟩ if it's equivalent to `\lst@g`⟨*name*⟩`@sty`.

```
    {\expandafter\ifx\csname\@lst#1@##1\endcsname#3%
        \global\expandafter\let\csname\@lst#1@##1\endcsname\relax
     \fi}}
```

Thanks to Magnus Lewis-Smith a wrong `#2` in the replacement text could be changed to `#3`.

And now memory-saving tests.

```
\fi
\lst@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.

```
\gdef\lst@IfOneOutOf#1\relax#2{%
    \def\lst@temp##1,#1,##2##3\relax{%
        \ifx\@empty##2\else \expandafter\lst@IOOOfirst \fi}%
    \def\lst@next{\lst@IfOneOutOf@#1\relax}%
    \expandafter\lst@next#2\relax\relax}
```

We either execute the ⟨*else*⟩ part or make the next test.

```
\gdef\lst@IfOneOutOf@#1\relax#2#3{%
    \ifx#2\relax
        \expandafter\@secondoftwo
    \else
        \expandafter\lst@temp\expandafter,#2,#1,\@empty\relax
        \expandafter\lst@next
    \fi}
\ifx\iffalse\else\fi
\gdef\lst@IOOOfirst#1\relax#2#3{\fi#2}
```

The line `\ifx\iffalse\else\fi` balances the `\fi` inside `\lst@IOOOfirst`.

`\lst@IFONEOUTOF` As in `\lst@IFONEOF` we need two `\uppercase`s here.

```
\gdef\lst@IFONEOUTOF#1\relax#2{%
    \uppercase{\def\lst@temp##1,#1},##2##3\relax{%
        \ifx\@empty##2\else \expandafter\lst@IOOOfirst \fi}%
    \def\lst@next{\lst@IFONEOUTOF@#1\relax}%
    \expandafter\lst@next#2\relax}
\gdef\lst@IFONEOUTOF@#1\relax#2#3{%
    \ifx#2\relax
        \expandafter\@secondoftwo
    \else
        \uppercase
            {\expandafter\lst@temp\expandafter,#2,#1,\@empty\relax}%
        \expandafter\lst@next
    \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.

```
\gdef\lst@KWTest{%
    \begingroup \let\lst@UM\@empty
    \expandafter\xdef\expandafter\@gtempa\expandafter{\the\lst@token}%
    \endgroup
    \expandafter\lst@IfOneOutOf\@gtempa\relax}
```

`\lst@KeywordTest` `\lst@KEYWORDTEST` are fairly easy now. Note that we don't need `#1`=⟨*prefix*⟩ here.

```
\gdef\lst@KeywordTest#1#2#3{\lst@KWTest #2{\let\lst@thestyle#3}{}}
\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` `\lst@WORKINGTEST` Ditto.

```
\gdef\lst@WorkingTest#1#2#3{\lst@KWTest #2#3{}}
\global\let\lst@WORKINGTEST\lst@WorkingTest
```

```
\fi
```

`sensitive` is a switch, preset `true` every language selection.

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

We select case insensitive definitions if necessary.

```
\lst@AddToHook{Init}
    {\lst@ifsensitive\else
         \let\lst@KeywordTest\lst@KEYWORDTEST
         \let\lst@WorkingTest\lst@WORKINGTEST
         \let\lst@IfOneOutOf\lst@IFONEOUTOF
     \fi}
```

`\lst@MakeMacroUppercase` makes the contents of `#1` (if defined) upper case.

```
\gdef\lst@MakeMacroUppercase#1{%
    \ifx\@undefined#1\else \uppercase\expandafter
        {\expandafter\def\expandafter#1\expandafter{#1}}%
    \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.

```
\gdef\lst@InstallTest#1#2#3#4#5#6#7#8{%
    \lst@AddToHook{TrackKeywords}{\lst@TrackKeywords{#1}#2#4#6#7#8}%
    \lst@AddToHook{PostTrackKeywords}{\lst@PostTrackKeywords#2#3#4#5}}
```

```
\lst@AddToHook{Init}{\lsthk@TrackKeywords\lsthk@PostTrackKeywords}
\lst@AddToHook{TrackKeywords}
    {\global\let\lst@DoDefineKeywords\@empty}% init
\lst@AddToHook{PostTrackKeywords}
    {\lst@DoDefineKeywords
     \global\let\lst@DoDefineKeywords\@empty}% init
```

We have to detect the keywords somewhere.

```
\lst@AddToHook{Output}{\lst@ifkeywords \lsthk@DetectKeywords \fi}
\lst@AddToHook{DetectKeywords}{}% init
\lst@AddToHook{ModeTrue}{\let\lst@ifkeywords\iffalse}
\lst@AddToHookExe{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.

```
\gdef\lst@InstallTestNow#1#2#3#4#5{%
    \@ifundefined{\string#2#1}%
    {\global\@namedef{\string#2#1}{}%
     \edef\@tempa{%
         \noexpand\lst@AddToHook{\ifx#5dDetectKeywords\else Output\fi}%
         {\ifx #4w\noexpand\lst@WorkingTest
             \else\noexpand\lst@KeywordTest \fi
          {#1}\noexpand#2\noexpand#3}}%
```

If we are advised to save memory, we insert a test for each ⟨*name*⟩. Otherwise we install the tests according to ⟨*prefix*⟩.

```
     \lst@ifsavemem
         \@tempa
     \else
         \@ifundefined{\@lst#1@if@ins}%
             {\@tempa \global\@namedef{\@lst#1@if@ins}{}}%
             {}%
     \fi}
    {}}
```

`\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 `\relax`es as terminators. This will define the keywords of all the classes as keywords of type ⟨*prefix*⟩. More details come soon.

```
\gdef\lst@TrackKeywords#1#2#3#4#5#6{%
    \lst@false
    \def\lst@arg{{#1}#4}%
    \expandafter\expandafter\expandafter\lst@TK@
        \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.

```
    \lst@ifsavemem\else
        \def\lst@arg{{#1}#4#2}%
        \expandafter\expandafter\expandafter\lst@TK@@
            \expandafter\lst@arg#3\relax\relax
    \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.

```
    \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*⟩ conatin 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*⟩

Thanks to Holger Arndt the definition of keywords is now delayed via `\lst@DoDefineKeywords`.

```
\gdef\lst@TK@#1#2#3#4{%
  \ifx\lst@ifsensitive\lst@ifsensitivedefed
    \ifx#3#4\else
      \lst@true
      \lst@ifsavemem\else
          \lst@UndefineKeywords{#1}#4#2%
          \lst@AddTo\lst@DoDefineKeywords{\lst@DefineKeywords{#1}#3#2}%
      \fi
    \fi
  \else
    \ifx#3\relax\else
      \lst@true
      \lst@ifsavemem\else
          \lst@UndefineKeywords{#1}#4#2%
          \lst@AddTo\lst@DoDefineKeywords{\lst@DefineKeywords{#1}#3#2}%
      \fi
    \fi
  \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.

```
  \lst@ifsavemem \ifx#3\relax\else
      \lst@ifsensitive\else \lst@MakeMacroUppercase#3\fi
  \fi \fi
```

Reaching the end of the class list, we end the loop.

```
  \ifx#3\relax
      \expandafter\@gobblethree
  \fi
  \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*⟩

```
\gdef\lst@TK@@#1#2#3#4#5{%
    \ifx#4\relax
        \expandafter\@gobblefour
    \else
        \lst@IfSubstring{#4#5}#3{}{\lst@UndefineKeywords{#1}#5#2}%
    \fi
    \lst@TK@@{#1}#2#3}
```

Keywords are up-to-date after `InitVars`.

```
\lst@AddToHook{InitVars}
    {\global\let\lst@ifsensitivedefed\lst@ifsensitive}
```

`\lst@PostTrackKeywords` After updating all the keywords, the global keywords and the global list become equivalent to the local ones.

```
\gdef\lst@PostTrackKeywords#1#2#3#4{%
    \lst@ifsavemem\else
        \global\let#3#1%
        \global\let#4#2%
    \fi}
```

16.3 Classes and families

`classoffset` just stores the argument in a macro.

```
\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.

```
\gdef\lst@InstallFamily#1#2#3#4#5{%
    \lst@Key{#2}\relax{\lst@UseFamily{#2}##1\relax\lst@MakeKeywords}%
    \lst@Key{more#2}\relax
        {\lst@UseFamily{#2}##1\relax\lst@MakeMoreKeywords}%
    \lst@Key{delete#2}\relax
        {\lst@UseFamily{#2}##1\relax\lst@DeleteKeywords}%
    \ifx\@empty#3\@empty\else
```

```
        \lst@Key{#3}{#4}{\lstKV@OptArg[\@ne]{##1}%
            {\@tempcnta\lst@classoffset \advance\@tempcnta####1\relax
             \@namedef{lst@#3\ifnum\@tempcnta=\@ne\else \the\@tempcnta
                            \fi}{####2}}}%
    \fi
    \expandafter\lst@InstallFamily@
        \csname\@lst @#2@data\expandafter\endcsname
        \csname\@lst @#5\endcsname {#1}{#2}{#3}}
```

Now we check whether ⟨*working procedure*⟩ is empty. Accordingly we use `w`orking procedure or `s`tyle in the 'data' definition. The working procedure is defined right here if necessary.

```
\gdef\lst@InstallFamily@#1#2#3#4#5#6#7#8{%
    \gdef#1{{#3}{#4}{#5}#2#7}%
    \long\def\lst@temp##1{#6}%
    \ifx\lst@temp\@gobble
        \lst@AddTo#1{s#8}%
    \else
        \lst@AddTo#1{w#8}%
        \global\@namedef{lst@g#4@wp}##1{#6}%
    \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, ...

```
\gdef\lst@UseFamily#1{%
    \def\lst@family{#1}%
    \@ifnextchar[\lst@UseFamily@{\lst@UseFamily@[\@ne]}}
\gdef\lst@UseFamily@[#1]{%
    \@tempcnta\lst@classoffset \advance\@tempcnta#1\relax
    \lst@ProvideFamily\lst@family
```

... and build the control sequences ...

```
    \lst@UseFamily@a
        {\lst@family\ifnum\@tempcnta=\@ne\else \the\@tempcnta \fi}}
\gdef\lst@UseFamily@a#1{%
    \expandafter\lst@UseFamily@b
       \csname\@lst @#1@list\expandafter\endcsname
       \csname\@lst @#1\expandafter\endcsname
       \csname\@lst @#1@also\expandafter\endcsname
       \csname\@lst @g#1\endcsname}
```

... required for `\lst@MakeKeywords` and `#6`.

```
\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.

```
\gdef\lst@ProvideFamily#1{%
    \@ifundefined{lstfam@#1\ifnum\@tempcnta=\@ne\else\the\@tempcnta\fi}%
    {\global\@namedef{lstfam@#1\ifnum\@tempcnta=\@ne\else
                                       \the\@tempcnta\fi}{}%
     \expandafter\expandafter\expandafter\lst@ProvideFamily@
```

```
         \csname\@lst @#1@data\endcsname
         {\ifnum\@tempcnta=\@ne\else \the\@tempcnta \fi}}%
    {}}%
```

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.

```
\gdef\lst@ProvideFamily@#1#2#3#4#5#6#7#8{%
    \expandafter\xdef\csname\@lst @g#2#8@sty\endcsname
    {\if #6w%
         \expandafter\noexpand\csname\@lst @g#2@wp\endcsname{#8}%
     \else
         \expandafter\noexpand\csname\@lst @#3#8\endcsname
     \fi}%
```

We ensure the existence of the style macro. This is done in the `Init` hook by assigning the default style if necessary.

```
    \ifx\@empty#3\@empty\else
        \edef\lst@temp{\noexpand\lst@AddToHook{Init}{%
            \noexpand\lst@ProvideStyle\expandafter\noexpand
                \csname\@lst @#3#8\endcsname\noexpand#4}}%
        \lst@temp
    \fi
```

We call a submacro to do the rest. It requires some control sequences.

```
    \expandafter\lst@ProvideFamily@@
         \csname\@lst @#2#8@list\expandafter\endcsname
         \csname\@lst @#2#8\expandafter\endcsname
         \csname\@lst @#2#8@also\expandafter\endcsname
         \csname\@lst @g#2#8@list\expandafter\endcsname
         \csname\@lst @g#2#8\expandafter\endcsname
         \csname\@lst @g#2#8@sty\expandafter\endcsname
         {#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).

```
\gdef\lst@ProvideFamily@@#1#2#3#4#5#6#7#8{%
    \gdef#1{#2#5}\global\let#2\@empty \global\let#3\@empty % init
    \gdef#4{#2#5}\global\let#5\@empty % init
    \if #8l\relax
        \lst@AddToHook{SetLanguage}{\def#1{#2#5}\let#2\@empty}%
    \fi
    \lst@InstallTest{#7}#1#2#4#5#6}
```

`\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.

```
\gdef\lst@InstallKeywords#1#2#3#4#5{%
    \lst@Key{#2}\relax
        {\lst@UseFamily{#2}[\@ne]##1\relax\lst@MakeKeywords}%
    \lst@Key{more#2}\relax
        {\lst@UseFamily{#2}[\@ne]##1\relax\lst@MakeMoreKeywords}%
    \lst@Key{delete#2}\relax
        {\lst@UseFamily{#2}[\@ne]##1\relax\lst@DeleteKeywords}%
    \ifx\@empty#3\@empty\else
        \lst@Key{#3}{#4}{\@namedef{lst@#3}{##1}}%
    \fi
    \expandafter\lst@InstallFamily@
        \csname\@lst @#2@data\expandafter\endcsname
        \csname\@lst @#5\endcsname {#1}{#2}{#3}}
```

`\lst@ProvideStyle` If the style macro `#1` is not defined, it becomes equivalent to `#2`.

```
\gdef\lst@ProvideStyle#1#2{%
    \ifx#1\@undefined \let#1#2%
    \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`.

```
\gdef\lst@BuildClassList#1#2,{%
    \ifx\relax#2\@empty\else
        \ifx\@empty#2\@empty\else
            \lst@lExtend#1{\csname\@lst @#2\expandafter\endcsname
                           \csname\@lst @g#2\endcsname}%
        \fi
        \expandafter\lst@BuildClassList\expandafter#1
    \fi}
```

`\lst@DeleteClassesIn` deletes pairs of tokens, namely the arguments `#2#3` to the submacro.

```
\gdef\lst@DeleteClassesIn#1#2{%
    \expandafter\lst@DCI@\expandafter#1#2\relax\relax}
\gdef\lst@DCI@#1#2#3{%
    \ifx#2\relax
```

```
        \expandafter\@gobbletwo
    \else
```

If we haven't reached the end of the class list, we define a temporary macro which removes all appearances.

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

```
\gdef\lst@MakeKeywords[#1]#2#3#4#5#6{%
    \def#3{#4#6}\let#4\@empty \let#5\@empty
    \lst@MakeMoreKeywords[#1]{#2}#3#4#5#6}
```

`\lst@MakeMoreKeywords` We append classes and keywords.

```
\gdef\lst@MakeMoreKeywords[#1]#2#3#4#5#6{%
    \lst@BuildClassList#3#1,\relax,%
    \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.

```
\gdef\lst@DeleteKeywords[#1]#2#3#4#5#6{%
    \lst@MakeKeywords[#1]{#2}\@tempa\@tempb#5#6%
    \lst@DeleteClassesIn#3\@tempa
    \lst@DeleteKeysIn#4\@tempb}
```

16.4 Main families and classes

Keywords

`keywords` Defining the keyword family gets very, very easy.

```
\lst@InstallFamily k{keywords}{keywordstyle}\bfseries{keywordstyle}{}ld
```

The following macro sets a keywordstyle, which …

```
\gdef\lst@DefKeywordstyle#1#2\@nil@{%
    \@namedef{lst@keywordstyle\ifnum\@tempcnta=\@ne\else\the\@tempcnta
                             \fi}{#1#2}}%
```

. . . is put together here. If we detect a star after the class number, we insert code to make the keyword uppercase.

```
\lst@Key{keywordstyle}{\bfseries}{\lstKV@OptArg[\@ne]{#1}%
  {\@tempcnta\lst@classoffset \advance\@tempcnta##1\relax
   \@ifstar{\lst@DefKeywordstyle{\uppercase\expandafter{%
                                \expandafter\lst@token
                                \expandafter{\the\lst@token}}}}%
           {\lst@DefKeywordstyle{}}##2\@nil@}}
```

`ndkeywords` Second order keywords use the same trick as `\lst@InstallKeywords`.

```
\lst@Key{ndkeywords}\relax
    {\lst@UseFamily{keywords}[\tw@]#1\relax\lst@MakeKeywords}%
\lst@Key{morendkeywords}\relax
    {\lst@UseFamily{keywords}[\tw@]#1\relax\lst@MakeMoreKeywords}%
\lst@Key{deletendkeywords}\relax
    {\lst@UseFamily{keywords}[\tw@]#1\relax\lst@DeleteKeywords}%
\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.

```
\lst@Key{keywordsprefix}\relax{\lst@DefActive\lst@keywordsprefix{#1}}
\global\let\lst@keywordsprefix\@empty
\lst@AddToHook{SelectCharTable}
    {\ifx\lst@keywordsprefix\@empty\else
         \expandafter\lst@CArg\lst@keywordsprefix\relax
             \lst@CDef{}%
                      {\lst@ifletter\else
                           \global\let\lst@prefixkeyword\@empty
                       \fi}%
                      {}%
     \fi}
\lst@AddToHook{Init}{\global\let\lst@prefixkeyword\relax}
\lst@AddToHook{Output}
    {\ifx\lst@prefixkeyword\@empty
         \let\lst@thestyle\lst@gkeywords@sty
         \global\let\lst@prefixkeyword\relax
     \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.

```
\lst@Key{otherkeywords}{}{%
    \let\lst@otherkeywords\@empty
    \lst@for{#1}\do{%
      \lst@MakeActive{##1}%
      \lst@lExtend\lst@otherkeywords{%
          \expandafter\lst@CArg\lst@temp\relax\lst@CDef
              {}\lst@PrintOtherKeyword\@empty}}}
\lst@AddToHook{SelectCharTable}{\lst@otherkeywords}
```

`\lst@PrintOtherkeyword` has been changed to `\lst@PrintOtherKeyword` after a bug report by Peter Bartke.

`\lst@PrintOtherKeyword` print preceding characters, prepare the output and typeset the argument in keyword style. James Willans reported problems when the output routine is invoked within `\begingroup` and `\endgroup`. Now the definition is restructured.

```
\gdef\lst@PrintOtherKeyword#1\@empty{%
    \lst@XPrintToken
    \begingroup
      \lst@modetrue \lsthk@TextStyle
      \let\lst@ProcessDigit\lst@ProcessLetter
      \let\lst@ProcessOther\lst@ProcessLetter
      \lst@lettertrue
      #1%
  \lst@SaveToken
    \endgroup
\lst@RestoreToken
\global\let\lst@savedcurrstyle\lst@currstyle
\let\lst@currstyle\lst@gkeywords@sty
    \lst@Output
\let\lst@currstyle\lst@savedcurrstyle}
```

> To do: Which part of `TextStyle` hook is required? Is it required anymore, i.e.after the restruction? Need to move it elsewhere?

```
\lst@EndAspect
⟨/misc⟩
```

The emphasize family

is just one macro call here.

```
⟨*misc⟩
\lst@BeginAspect[keywords]{emph}
\lst@InstallFamily e{emph}{emphstyle}{}{emphstyle}{}od
\lst@EndAspect
⟨/misc⟩
```

TeX control sequences

Here we check the last ‘other’ processed token.

```
⟨*misc⟩
\lst@BeginAspect[keywords]{tex}
```

```
\lst@InstallFamily {cs}{texcs}{texcsstyle}\relax{keywordstyle}
    {\ifx\lst@lastother\lstum@backslash
         \expandafter\let\expandafter\lst@thestyle
                         \csname lst@texcsstyle#1\endcsname
     \fi}
    ld
```

The style-key checks for the optional star (which must be in front of the optional class argument).

```
\lst@Key{texcsstyle}\relax
  {\@ifstar{\lst@true\lst@DefTexcsstyle}%
           {\lst@false\lst@DefTexcsstyle}#1\@nil@}
```

```
\gdef\lst@DefTexcsstyle#1\@nil@{%
    \let\lst@iftexcsincludebs\lst@if
    \lstKV@OptArg[\@ne]{#1}%
    {\@tempcnta\lst@classoffset \advance\@tempcnta##1\relax
     \@namedef{lst@texcsstyle\ifnum\@tempcnta=\@ne\else
                                  \the\@tempcnta \fi}{##2}}}%
\global\let\lst@iftexcsincludebs\iffalse
```

To make the backslash belong to the control sequence, it is merged with the following token. This option was suggested by Morten Høgholm. Christian Schneider pointed out that the original implementation was broken when the identifier was preceded by an "other" character. To fix this (and other bugs), we first output whatever is in the current token before merging.

```
\let\lst@iftexcsincludebs\iffalse
\lst@AddToHook{SelectCharTable}
{\lst@iftexcsincludebs \ifx\@empty\lst@texcs\else
     \lst@DefSaveDef{`\\}\lsts@texcsbs
      {\lst@ifletter
           \lst@Output
       \else
           \lst@OutputOther
       \fi
       \lst@Merge\lsts@texcsbs}%
 \fi \fi}
```

```
\lst@EndAspect
⟨/misc⟩
```

Compiler directives

First some usual stuff.

directives

```
⟨*misc⟩
\lst@BeginAspect[keywords]{directives}
```

The initialization of `\lst@directives` has been added after a bug report from Kris Luyten.

```
\lst@NewMode\lst@CDmode
\lst@AddToHook{EOL}{\ifnum\lst@mode=\lst@CDmode \lst@LeaveMode \fi}
\lst@InstallKeywords{d}{directives}{directivestyle}\relax{keywordstyle}
    {\ifnum\lst@mode=\lst@CDmode
         \let\lst@thestyle\lst@directivestyle
     \fi}
    ld
\global\let\lst@directives\@empty % init
```

Now we define a new delimiter for directives: We enter 'directive mode' only in the first column.

```
\lst@AddTo\lst@delimtypes{,directive}
\gdef\lst@Delim@directive#1\@empty#2#3#4{%
    \lst@CArg #1\relax\lst@DefDelimB
        {\lst@CalcColumn}%
        {}%
        {\ifnum\@tempcnta=\z@
             \def\lst@bnext{#2\lst@CDmode{#4\lst@Lmodetrue}%
                \let\lst@currstyle\lst@directivestyle}%
 \fi
 \@gobblethree}%
        #2\lst@CDmode{#4\lst@Lmodetrue}}
```

We introduce a new string type (thanks to R. Isernhagen), which …

```
\lst@AddTo\lst@stringtypes{,directive}
\gdef\lst@StringDM@directive#1#2#3\@empty{%
   \lst@CArg #2\relax\lst@CDef
       {}%
```

… is active only in `\lst@CDmode`:

```
       {\let\lst@bnext\lst@CArgEmpty
        \ifnum\lst@mode=\lst@CDmode
            \def\lst@bnext{\lst@BeginString{#1}}%
        \fi
        \lst@bnext}%
       \@empty
   \lst@CArg #3\relax\lst@CDef
       {}%
       {\let\lst@enext\lst@CArgEmpty
        \ifnum #1=\lst@mode
            \let\lst@bnext\lst@EndString
        \fi
        \lst@bnext}%
       \@empty}
```

```
\lst@EndAspect
⟨/misc⟩
```

16.5 Keyword comments

includes both comment types and is possibly split into this and `dkcs`.

```
⟨*misc⟩
\lst@BeginAspect[keywords,comments]{keywordcomments}
```

`\lst@BeginKC`
`\lst@BeginKCS`

Starting a keyword comment is easy, but: (1) The submacros are called outside of two group levels, and …

```
\lst@NewMode\lst@KCmode \lst@NewMode\lst@KCSmode
\gdef\lst@BeginKC{\aftergroup\aftergroup\aftergroup\lst@BeginKC@}%
\gdef\lst@BeginKC@{%
   \lst@ResetToken
   \lst@BeginComment\lst@KCmode{{\lst@commentstyle}\lst@modetrue}%
                   \@empty}%
\gdef\lst@BeginKCS{\aftergroup\aftergroup\aftergroup\lst@BeginKCS@}%
\gdef\lst@BeginKCS@{%
   \lst@ResetToken
   \lst@BeginComment\lst@KCSmode{{\lst@commentstyle}\lst@modetrue}%
                   \@empty}%
```

(2) we must ensure that the comment starts after printing the comment delimiter since it could be a keyword. We assign `\lst@BeginKC`[S] to `\lst@KCpost`, which is executed and reset in `PostOutput`.

```
\lst@AddToHook{PostOutput}{\lst@KCpost \global\let\lst@KCpost\@empty}
\global\let\lst@KCpost\@empty % init
```

`\lst@EndKC` leaves the comment mode before the (temporaryly saved) comment delimiter is printed.

```
\gdef\lst@EndKC{\lst@SaveToken \lst@LeaveMode \lst@RestoreToken
    \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.

```
\lst@InstallKeywords{kc}{keywordcomment}{}\relax{}
    {\ifnum\lst@mode=\lst@KCmode
         \edef\lst@temp{\the\lst@token}%
         \ifx\lst@temp\lst@KCmatch
             \lst@EndKC
         \fi
     \else
         \lst@ifmode\else
             \xdef\lst@KCmatch{\the\lst@token}%
             \global\let\lst@KCpost\lst@BeginKC
         \fi
     \fi}
    lo
```

`keywordcommentsemicolon` The key simply stores the keywords. After a bug report by Norbert Eisinger the initialization in `SetLanguage` has been added.

```
\lst@Key{keywordcommentsemicolon}{}{\lstKV@ThreeArg{#1}%
    {\def\lst@KCAkeywordsB{##1}%
     \def\lst@KCAkeywordsE{##2}%
     \def\lst@KCBkeywordsB{##3}%
     \def\lst@KCkeywords{##1##2##3}}}
\lst@AddToHook{SetLanguage}{%
    \let\lst@KCAkeywordsB\@empty \let\lst@KCAkeywordsE\@empty
    \let\lst@KCBkeywordsB\@empty \let\lst@KCkeywords\@empty}
```

We define an appropriate semicolon if this keyword comment type is defined. Appropriate means that we leave any keyword comment mode if active. Oldrich Jedlicka reported a bug and provided the fix, the two `\@empty`s.

```
\lst@AddToHook{SelectCharTable}
    {\ifx\lst@KCkeywords\@empty\else
        \lst@DefSaveDef{`\;}\lsts@EKC
            {\lst@XPrintToken
             \ifnum\lst@mode=\lst@KCmode \lst@EndComment\@empty \else
             \ifnum\lst@mode=\lst@KCSmode \lst@EndComment\@empty
             \fi \fi
             \lsts@EKC}%
     \fi}
```

The 'working identifier' macros enter respectively leave comment mode.

```
\gdef\lst@KCAWorkB{%
    \lst@ifmode\else \global\let\lst@KCpost\lst@BeginKC \fi}
\gdef\lst@KCBWorkB{%
    \lst@ifmode\else \global\let\lst@KCpost\lst@BeginKCS \fi}
\gdef\lst@KCAWorkE{\ifnum\lst@mode=\lst@KCmode \lst@EndKC \fi}
```

Now we install the tests and initialize the given macros.

```
\lst@ProvideFamily@@
    \lst@KCAkeywordsB@list\lst@KCAkeywordsB \lst@KC@also
    \lst@gKCAkeywordsB@list\lst@gKCAkeywordsB \lst@KCAWorkB
    {kcb}owo % prefix, other key, working procedure, Output hook
\lst@ProvideFamily@@
    \lst@KCAkeywordsE@list\lst@KCAkeywordsE \lst@KC@also
    \lst@gKCAkeywordsE@list\lst@gKCAkeywordsE \lst@KCAWorkE
    {kce}owo
\lst@ProvideFamily@@
    \lst@KCBkeywordsB@list\lst@KCBkeywordsB \lst@KC@also
    \lst@gKCBkeywordsB@list\lst@gKCBkeywordsB \lst@KCBWorkB
    {kcs}owo

\lst@EndAspect
⟨/misc⟩
```

16.6 Export of identifiers

One more ‘keyword’ class.

`\lstindexmacro`

```
⟨*misc⟩
\lst@BeginAspect[keywords]{index}
\lst@InstallFamily w{index}{indexstyle}\lstindexmacro{indexstyle}
    {\csname\@lst @indexstyle#1\expandafter\endcsname
         \expandafter{\the\lst@token}}
    od
\lst@UserCommand\lstindexmacro#1{\index{{\ttfamily#1}}}
\lst@EndAspect
⟨/misc⟩
```

`procnamestyle`
`procnamekeys`
`indexprocnames`

The ‘idea’ here is the usage of a global `\lst@ifprocname`, indicating a preceding ‘procedure keyword’. All the other is known stuff.

```
⟨*misc⟩
\lst@BeginAspect[keywords]{procnames}
\gdef\lst@procnametrue{\global\let\lst@ifprocname\iftrue}
\gdef\lst@procnamefalse{\global\let\lst@ifprocname\iffalse}
\lst@AddToHook{Init}{\lst@procnamefalse}
\lst@AddToHook{DetectKeywords}
    {\lst@ifprocname
        \let\lst@thestyle\lst@procnamestyle
        \lst@ifindexproc \csname\@lst @gindex@sty\endcsname \fi
        \lst@procnamefalse
     \fi}
```

And these are the two implemented keys:

```
\lst@Key{procnamestyle}{}{\def\lst@procnamestyle{#1}}
\lst@Key{indexprocnames}{false}[t]{\lstKV@SetIf{#1}\lst@ifindexproc}
\lst@AddToHook{Init}{\lst@ifindexproc \lst@indexproc \fi}
\gdef\lst@indexproc{%
    \@ifundefined{lst@indexstyle1}%
        {\@namedef{lst@indexstyle1}##1{}}%
        {}}
```

The default definition of `\lst@indexstyle` above has been moved outside the hook after a bug report from Ulrich G. Wortmann.

```
\lst@InstallKeywords w{procnamekeys}{}\relax{}
    {\global\let\lst@PNpost\lst@procnametrue}
    od
\lst@AddToHook{PostOutput}{\lst@PNpost\global\let\lst@PNpost\@empty}
\global\let\lst@PNpost\@empty % init
\lst@EndAspect
⟨/misc⟩
```

17 More aspects and keys

`basicstyle`
`inputencoding`

There is no better place to define these keys, I think.

```
⟨*kernel⟩
\lst@Key{basicstyle}\relax{\def\lst@basicstyle{#1}}
\lst@Key{inputencoding}\relax{\def\lst@inputenc{#1}}
\lst@AddToHook{Init}
    {\lst@basicstyle
     \ifx\lst@inputenc\@empty\else
         \@ifundefined{inputencoding}{}%
            {\inputencoding\lst@inputenc}%
     \fi}
\lst@AddToHookExe{EmptyStyle}
    {\let\lst@basicstyle\@empty
     \let\lst@inputenc\@empty}
\lst@Key{multicols}{}{\@tempcnta=0#1\relax\def\lst@multicols{#1}}
⟨/kernel⟩
```

Michael Niedermair asked for a key like `inputencoding`.

17.1 Styles and languages

We begin with style definition and selection.

```
⟨*misc⟩
\lst@BeginAspect{style}
```

`\lststylefiles` This macro is defined if and only if it's undefined yet.

```
\@ifundefined{lststylefiles}
    {\lst@UserCommand\lststylefiles{lststy0.sty}}{}
```

`\lstdefinestyle`
`\lst@definestyle`
`\lst@DefStyle`

are defined in terms of `\lst@DefStyle`, which is defined via `\lst@DefDriver`.

```
\lst@UserCommand\lstdefinestyle{\lst@DefStyle\iftrue}
\lst@UserCommand\lst@definestyle{\lst@DefStyle\iffalse}
\gdef\lst@DefStyle{\lst@DefDriver{style}{sty}\lstset}
```

The 'empty' style calls the initial empty hook `EmptyStyle`.

```
\global\@namedef{lststy@$}{\lsthk@EmptyStyle}
\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.

```
\lst@Key{style}\relax{%
```

```
    \lst@LAS{style}{sty}{[]{#1}}\lst@NoAlias\lststylefiles
        \lsthk@SetStyle
        {}}
```

```
\lst@AddToHook{SetStyle}{}% init
```

```
\lst@EndAspect
⟨/misc⟩
```

Now we deal with commands used in defining and selecting programming languages, in particular with aliases.

```
⟨*misc⟩
\lst@BeginAspect{language}
```

`\lstlanguagefiles` This macro is defined if and only if it's undefined yet.

```
\@ifundefined{lstdriverfiles}
    {\lst@UserCommand\lstlanguagefiles{lstlang0.sty}}{}
```

`\lstdefinelanguage`
`\lst@definelanguage`
`\lst@DefLang`

are defined in terms of `\lst@DefLang`, which is defined via `\lst@DefDriver`.

```
\lst@UserCommand\lstdefinelanguage{\lst@DefLang\iftrue}
\lst@UserCommand\lst@definelanguage{\lst@DefLang\iffalse}
\gdef\lst@DefLang{\lst@DefDriver{language}{lang}\lstset}
```

Now we can provide the 'empty' language.

```
\lstdefinelanguage{}{}
```

`language`
`alsolanguage`

is mainly an application of `\lst@LAS`.

```
\lst@Key{language}\relax{\lstKV@OptArg[]{#1}%
    {\lst@LAS{language}{lang}{[##1]{##2}}\lst@FindAlias\lstlanguagefiles
         \lsthk@SetLanguage
         {\lst@FindAlias[##1]{##2}%
          \let\lst@language\lst@malias
          \let\lst@dialect\lst@oalias}}}
```

Ditto, we simply don't execute `\lsthk@SetLanguage`.

```
\lst@Key{alsolanguage}\relax{\lstKV@OptArg[]{#1}%
    {\lst@LAS{language}{lang}{[##1]{##2}}\lst@FindAlias\lstlanguagefiles
         {}%
         {\lst@FindAlias[##1]{##2}%
          \let\lst@language\lst@malias
          \let\lst@dialect\lst@oalias}}}
```

```
\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. Thanks to Walter E. Brown for reporting a problem with the argument delimiter '[' in a previous definition of `\lstalias@`.

```
\lst@UserCommand\lstalias{\@ifnextchar[\lstalias@\lstalias@@}
\gdef\lstalias@[#1]#2{\lstalias@b #2$#1}
\gdef\lstalias@b#1[#2]#3{\lst@NormedNameDef{lsta@#1}{#3$#2}}
\gdef\lstalias@@#1#2{\lst@NormedNameDef{lsta@#1}{#2}}
```

`defaultdialect` We simply store the dialect.

```
\lst@Key{defaultdialect}\relax
    {\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.

```
\gdef\lst@FindAlias[#1]#2{%
    \lst@NormedDef\lst@oalias{#1}%
    \lst@NormedDef\lst@malias{#2}%
    \@ifundefined{lsta@\lst@malias}{}%
        {\edef\lst@malias{\csname\@lst a@\lst@malias\endcsname}}%

    \ifx\@empty\lst@oalias \@ifundefined{lstdd@\lst@malias}{}%
        {\edef\lst@oalias{\csname\@lst dd@\lst@malias\endcsname}}%
    \fi
```

Now we are ready for an alias of a single dialect.

```
    \edef\lst@temp{\lst@malias $\lst@oalias}%
    \@ifundefined{lsta@\lst@temp}{}%
        {\edef\lst@temp{\csname\@lst a@\lst@temp\endcsname}}%
```

Finally we again set the default dialect—for the case of a dialect alias.

```
    \expandafter\lst@FindAlias@\lst@temp $}
\gdef\lst@FindAlias@#1$#2${%
    \def\lst@malias{#1}\def\lst@oalias{#2}%
    \ifx\@empty\lst@oalias \@ifundefined{lstdd@\lst@malias}{}%
        {\edef\lst@oalias{\csname\@lst dd@\lst@malias\endcsname}}%
    \fi}
```

`\lst@RequireLanguages` This definition will be equivalent to `\lstloadlanguages`. We requested the given list of languages and load additionally required aspects.

```
\gdef\lst@RequireLanguages#1{%
    \lst@Require{language}{lang}{#1}\lst@FindAlias\lstlanguagefiles
    \ifx\lst@loadaspects\@empty\else
        \lst@RequireAspects\lst@loadaspects
    \fi}
```

`\lstloadlanguages` is the same as `\lst@RequireLanguages`.

```
\global\let\lstloadlanguages\lst@RequireLanguages

\lst@EndAspect
⟨/misc⟩
```

17.2 Format definitions*

```
⟨*misc⟩
\lst@BeginAspect{formats}
```

`\lstformatfiles` This macro is defined if and only if it's undefined yet.

```
\@ifundefined{lstformatfiles}
    {\lst@UserCommand\lstformatfiles{lstfmt0.sty}}{}
```

`\lstdefineformat` `\lst@defineformat` `\lst@DefFormat` are defined in terms of `\lst@DefFormat`, which is defined via `\lst@DefDriver`.

```
\lst@UserCommand\lstdefineformat{\lst@DefFormat\iftrue}
\lst@UserCommand\lst@defineformat{\lst@DefFormat\iffalse}
\gdef\lst@DefFormat{\lst@DefDriver{format}{fmt}\lst@UseFormat}
```

We provide the ‘empty’ format.

```
\lstdefineformat{}{}
```

`format` is an application of `\lst@LAS`. We just specify the hook as ‘pre’ and an empty argument as ‘post’ code.

```
\lst@Key{format}\relax{%
    \lst@LAS{format}{fmt}{[]{#1}}\lst@NoAlias\lstformatfiles
        \lsthk@SetFormat
        {}}
```

```
\lst@AddToHook{SetFormat}{\let\lst@fmtformat\@empty}% init
```

Helpers Our goal is to define the yet unkown `\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`.

```
\gdef\lst@fmtSplit#1#2{%
    \def\lst@temp##1#2##2\relax##3{%
        \ifnum##3=\z@
            \ifx\@empty##2\@empty
                \lst@false
                \let\lst@fmta#1%
                \let\lst@fmtb\@empty
            \else
                \expandafter\lst@temp#1\relax\@ne
            \fi
        \else
            \def\lst@fmta{##1}\def\lst@fmtb{##2}%
        \fi}%
    \lst@true
    \expandafter\lst@temp#1#2\relax\z@}
```

`\lst@IfNextCharWhitespace` is defined in terms of `\lst@IfSubstring`.

```
\gdef\lst@IfNextCharWhitespace#1#2#3{%
    \lst@IfSubstring#3\lst@whitespaces{#1}{#2}#3}
```

And here come all white space characters.

```
\begingroup
\catcode`\^^I=12\catcode`\^^J=12\catcode`\^^M=12\catcode`\^^L=12\relax%
\lst@DefActive\lst@whitespaces{\ ^^I^^J^^M}% add ^^L
\global\let\lst@whitespaces\lst@whitespaces%
\endgroup
```

`\lst@fmtIfIdentifier` tests the first character of `#1`

```
\gdef\lst@fmtIfIdentifier#1{%
    \ifx\relax#1\@empty
        \expandafter\@secondoftwo
    \else
        \expandafter\lst@fmtIfIdentifier@\expandafter#1%
    \fi}
```

against the ‘letters’ `_`, `@`, `A`,. . . ,`Z` and `a`,. . . ,`z`.

```
\gdef\lst@fmtIfIdentifier@#1#2\relax{%
    \let\lst@next\@secondoftwo
    \ifnum`#1=`_\else
    \ifnum`#1<64\else
    \ifnum`#1<91\let\lst@next\@firstoftwo\else
    \ifnum`#1<97\else
    \ifnum`#1<123\let\lst@next\@firstoftwo\else
    \fi \fi \fi \fi \fi
    \lst@next}
```

`\lst@fmtIfNextCharIn` is required for the optional ⟨*exceptional characters*⟩. The implementation is easy—refer section 13.1.

```
\gdef\lst@fmtIfNextCharIn#1{%
    \ifx\@empty#1\@empty \expandafter\@secondoftwo \else
                         \def\lst@next{\lst@fmtIfNextCharIn@{#1}}%
                         \expandafter\lst@next\fi}
\gdef\lst@fmtIfNextCharIn@#1#2#3#4{%
    \def\lst@temp##1#4##2##3\relax{%
        \ifx \@empty##2\expandafter\@secondoftwo
                 \else \expandafter\@firstoftwo \fi}%
    \lst@temp#1#4\@empty\relax{#2}{#3}#4}
```

`\lst@fmtCDef` We need derivations of `\lst@CDef` and `\lst@CDefX`: we have to test the next character against the sequence `#5` of exceptional characters. These tests are inserted here.

```
\gdef\lst@fmtCDef#1{\lst@fmtCDef@#1}
\gdef\lst@fmtCDef@#1#2#3#4#5#6#7{%
    \lst@CDefIt#1{#2}{#3}%
               {\lst@fmtIfNextCharIn{#5}{#4#2#3}{#6#4#2#3#7}}%
               #4%
               {}{}{}}
```

`\lst@fmtCDefX` The same but ‘drop input’.

```
\gdef\lst@fmtCDefX#1{\lst@fmtCDefX@#1}
\gdef\lst@fmtCDefX@#1#2#3#4#5#6#7{%
    \let#4#1%
    \ifx\@empty#2\@empty
        \def#1{\lst@fmtIfNextCharIn{#5}{#4}{#6#7}}%
    \else \ifx\@empty#3\@empty
        \def#1##1{%
            \ifx##1#2%
                \def\lst@next{\lst@fmtIfNextCharIn{#5}{#4##1}%
                                                      {#6#7}}%
            \else
                 \def\lst@next{#4##1}%
            \fi
            \lst@next}%
    \else
        \def#1{%
```

```
            \lst@IfNextCharsArg{#2#3}%
                {\lst@fmtIfNextCharIn{#5}{\expandafter#4\lst@eaten}%
                                          {#6#7}}%
                {\expandafter#4\lst@eaten}}%
    \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.

```
\gdef\lst@UseFormat#1{%
    \def\lst@fmtwhole{#1}%
    \lst@UseFormat@}
\gdef\lst@UseFormat@{%
    \lst@fmtSplit\lst@fmtwhole,%
```

We assign the rest of the format definition, …

```
    \let\lst@fmtwhole\lst@fmtb
    \ifx\lst@fmta\@empty\else
```

… split the item at the equal sign, and work on the item.

```
        \lst@fmtSplit\lst@fmta=%
        \ifx\@empty\lst@fmta\else
```

To do: Insert `\let\lst@arg\@empty \expandafter\lst@XConvert\lst@fmtb\@nil \let\lst@fmtb\lst@arg`.

```
            \expandafter\lstKV@XOptArg\expandafter[\expandafter]%
                \expandafter{\lst@fmtb}\lst@UseFormat@b
        \fi
    \fi
```

Finally we process the next item if the rest is not empty.

```
    \ifx\lst@fmtwhole\@empty\else
        \expandafter\lst@UseFormat@
    \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`.

```
\gdef\lst@UseFormat@b[#1]#2{%
    \def\lst@fmtc{{#1}}\lst@lExtend\lst@fmtc{\expandafter{\lst@fmta}}%
    \def\lst@fmtb{#2}%
    \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`.

```
    \ifx\@empty\lst@fmta
        \lst@lAddTo\lst@fmtc{{}}%
    \else
        \lst@lExtend\lst@fmtc{\expandafter
            {\expandafter\lst@fmtPre\expandafter{\lst@fmta}}}%
    \fi
    \ifx\@empty\lst@fmtb
        \lst@lAddTo\lst@fmtc{{}}%
    \else
        \lst@lExtend\lst@fmtc{\expandafter
            {\expandafter\lst@fmtPost\expandafter{\lst@fmtb}}}%
    \fi
```

Eventually we extend `\lst@fmtformat` appropriately. Note that `\lst@if` still indicates whether the replacement tokens contain `\string`.

```
    \expandafter\lst@UseFormat@c\lst@fmtc}
```

```
\gdef\lst@UseFormat@c#1#2#3#4{%
    \lst@fmtIfIdentifier#2\relax
    {\lst@fmtIdentifier{#2}%
     \lst@if\else \PackageWarning{Listings}%
         {Cannot drop identifier in format definition}%
     \fi}%
    {\lst@if
         \lst@lAddTo\lst@fmtformat{\lst@CArgX#2\relax\lst@fmtCDef}%
     \else
         \lst@lAddTo\lst@fmtformat{\lst@CArgX#2\relax\lst@fmtCDefX}%
     \fi
     \lst@DefActive\lst@fmtc{#1}%
     \lst@lExtend\lst@fmtformat{\expandafter{\lst@fmtc}{#3}{#4}}}}
```

```
\lst@AddToHook{SelectCharTable}{\lst@fmtformat}
\global\let\lst@fmtformat\@empty
```

The formatting

`\lst@fmtPre`

```
\gdef\lst@fmtPre#1{%
    \lst@PrintToken
    \begingroup
    \let\newline\lst@fmtEnsureNewLine
    \let\space\lst@fmtEnsureSpace
    \let\indent\lst@fmtIndent
    \let\noindent\lst@fmtNoindent
    #1%
    \endgroup}
```

`\lst@fmtPost`

```
\gdef\lst@fmtPost#1{%
    \global\let\lst@fmtPostOutput\@empty
    \begingroup
    \def\newline{\lst@AddTo\lst@fmtPostOutput\lst@fmtEnsureNewLine}%
    \def\space{\aftergroup\lst@fmtEnsurePostSpace}%
    \def\indent{\lst@AddTo\lst@fmtPostOutput\lst@fmtIndent}%
    \def\noindent{\lst@AddTo\lst@fmtPostOutput\lst@fmtNoindent}%
    \aftergroup\lst@PrintToken
    #1%
    \endgroup}
```

```
\lst@AddToHook{Init}{\global\let\lst@fmtPostOutput\@empty}
\lst@AddToHook{PostOutput}
    {\lst@fmtPostOutput \global\let\lst@fmtPostOutput\@empty}
```

`\lst@fmtEnsureSpace`
`\lst@fmtEnsurePostSpace`

```
\gdef\lst@fmtEnsureSpace{%
    \lst@ifwhitespace\else \expandafter\lst@ProcessSpace \fi}
\gdef\lst@fmtEnsurePostSpace{%
    \lst@IfNextCharWhitespace{}{\lst@ProcessSpace}}
```

fmtindent

\lst@fmtIndent

\lst@fmtNoindent

```
\lst@Key{fmtindent}{20pt}{\def\lst@fmtindent{#1}}
\newdimen\lst@fmtcurrindent
\lst@AddToHook{InitVars}{\global\lst@fmtcurrindent\z@}
\gdef\lst@fmtIndent{\global\advance\lst@fmtcurrindent\lst@fmtindent}
\gdef\lst@fmtNoindent{\global\advance\lst@fmtcurrindent-\lst@fmtindent}
```

\lst@fmtEnsureNewLine

```
\gdef\lst@fmtEnsureNewLine{%
    \global\advance\lst@newlines\@ne
    \global\advance\lst@newlinesensured\@ne
    \lst@fmtignoretrue}
```

```
\lst@AddToAtTop\lst@DoNewLines{%
    \ifnum\lst@newlines>\lst@newlinesensured
        \global\advance\lst@newlines-\lst@newlinesensured
    \fi
    \global\lst@newlinesensured\z@}
\newcount\lst@newlinesensured % global
\lst@AddToHook{Init}{\global\lst@newlinesensured\z@}
```

```
\gdef\lst@fmtignoretrue{\let\lst@fmtifignore\iftrue}
\gdef\lst@fmtignorefalse{\let\lst@fmtifignore\iffalse}
\lst@AddToHook{InitVars}{\lst@fmtignorefalse}
\lst@AddToHook{Output}{\lst@fmtignorefalse}
```

\lst@fmtUseLostSpace

```
\gdef\lst@fmtUseLostSpace{%
    \lst@ifnewline \kern\lst@fmtcurrindent \global\lst@lostspace\z@
    \else
        \lst@OldOLS
    \fi}
\lst@AddToHook{Init}
    {\lst@true
     \ifx\lst@fmtformat\@empty \ifx\lst@fmt\@empty \lst@false \fi\fi
     \lst@if
        \let\lst@OldOLS\lst@OutputLostSpace
        \let\lst@OutputLostSpace\lst@fmtUseLostSpace
        \let\lst@ProcessSpace\lst@fmtProcessSpace
     \fi}
```

To do: This ‘lost space’ doesn’t use `\lst@alloverstyle` yet!

\lst@fmtProcessSpace

```
\gdef\lst@fmtProcessSpace{%
    \lst@ifletter
        \lst@Output
        \lst@fmtifignore\else
            \lst@AppendOther\lst@outputspace
        \fi
    \else \lst@ifkeepspaces
        \lst@AppendOther\lst@outputspace
    \else \ifnum\lst@newlines=\z@
        \lst@AppendSpecialSpace
    \else \ifnum\lst@length=\z@
            \global\advance\lst@lostspace\lst@width
            \global\advance\lst@pos\m@ne
        \else
            \lst@AppendSpecialSpace
        \fi
    \fi \fi \fi
    \lst@whitespacetrue}
```

Formatting identifiers

`\lst@fmtIdentifier` We install a (keyword) test for the ‘format identifiers’.

```
\lst@InstallTest{f}
    \lst@fmt@list\lst@fmt \lst@gfmt@list\lst@gfmt
    \lst@gfmt@wp
    wd
\gdef\lst@fmt@list{\lst@fmt\lst@gfmt}\global\let\lst@fmt\@empty
\gdef\lst@gfmt@list{\lst@fmt\lst@gfmt}\global\let\lst@gfmt\@empty
```

The working procedure expands `\lst@fmt$`⟨*string*⟩ (and defines `\lst@PrintToken` to do nothing).

```
\gdef\lst@gfmt@wp{%
    \begingroup \let\lst@UM\@empty
    \let\lst@PrintToken\@empty
    \csname\@lst @fmt$\the\lst@token\endcsname
    \endgroup}
```

This control sequence is probably defined as ‘working identifier’.

```
\gdef\lst@fmtIdentifier#1#2#3#4{%
    \lst@DefOther\lst@fmta{#2}\edef\lst@fmt{\lst@fmt,\lst@fmta}%
    \@namedef{\@lst @fmt$\lst@fmta}{#3#4}}
```

`\lst@fmt$`⟨*identifier*⟩ expands to a `\lst@fmtPre`/`\lst@fmtPost` sequence defined by `#2` and `#3`.

```
\lst@EndAspect
⟨/misc⟩
```

17.3 Line numbers

Rolf Niepraschk asked for line numbers.

```
⟨*misc⟩
\lst@BeginAspect{labels}
```

`numbers` Depending on the argument we define `\lst@PlaceNumber` to print the line number.

```
\lst@Key{numbers}{none}{%
    \let\lst@PlaceNumber\@empty
    \lstKV@SwitchCases{#1}%
    {none:\\%
     left:\def\lst@PlaceNumber{\llap{\normalfont
                \lst@numberstyle{\thelstnumber}\kern\lst@numbersep}}\\%
     right:\def\lst@PlaceNumber{\rlap{\normalfont
               \kern\linewidth \kern\lst@numbersep
               \lst@numberstyle{\thelstnumber}}}%
    }{\PackageError{Listings}{Numbers #1 unknown}\@ehc}}
```

`numberstyle`
`numbersep`
`stepnumber`
`numberblanklines`
`numberfirstline`

Definition of the keys.

```
\lst@Key{numberstyle}{}{\def\lst@numberstyle{#1}}
\lst@Key{numbersep}{10pt}{\def\lst@numbersep{#1}}
\lst@Key{stepnumber}{1}{\def\lst@stepnumber{#1\relax}}
\lst@AddToHook{EmptyStyle}{\let\lst@stepnumber\@ne}
```

```
\lst@Key{numberblanklines}{true}[t]
    {\lstKV@SetIf{#1}\lst@ifnumberblanklines}
\lst@Key{numberfirstline}{f}[t]{\lstKV@SetIf{#1}\lst@ifnumberfirstline}
\gdef\lst@numberfirstlinefalse{\let\lst@ifnumberfirstline\iffalse}
```

`firstnumber`

We select the first number according to the argument.

```
\lst@Key{firstnumber}{auto}{%
    \lstKV@SwitchCases{#1}%
    {auto:\let\lst@firstnumber\@undefined\\%
     last:\let\lst@firstnumber\c@lstnumber
    }{\def\lst@firstnumber{#1\relax}}}
\lst@AddToHook{PreSet}{\let\lst@advancenumber\z@}
```

`\lst@firstnumber` is now set to `\lst@lineno` instead of `\lst@firstline`, as per changes in `lstpatch.sty` from 1.3b pertaining to linerange markers.

```
\lst@AddToHook{PreInit}
    {\ifx\lst@firstnumber\@undefined
         \def\lst@firstnumber{\lst@lineno}%
     \fi}
```

`\lst@SetFirstNumber`
`\lst@SaveFirstNumber`

Boris Veytsman proposed to continue line numbers according to listing names. We define the label number of the first printing line here. A bug reported by Jens Schwarzer has been removed by replacing `\@ne` by `\lst@firstline`.

```
\gdef\lst@SetFirstNumber{%
    \ifx\lst@firstnumber\@undefined
        \@tempcnta 0\csname\@lst no@\lst@intname\endcsname\relax
        \ifnum\@tempcnta=\z@ \@tempcnta\lst@firstline
                       \else \lst@nololtrue \fi
        \advance\@tempcnta\lst@advancenumber
        \edef\lst@firstnumber{\the\@tempcnta\relax}%
    \fi}
```

The current label is stored in`\lstno@`⟨*name*⟩. If the name is empty, we use a space instead, which leaves `\lstno@` undefined.

```
\gdef\lst@SaveFirstNumber{%
    \expandafter\xdef
        \csname\@lst no\ifx\lst@intname\@empty @ \else @\lst@intname\fi
        \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.

```
\newcounter{lstnumber}% \global
\global\c@lstnumber\@ne % init
\renewcommand*\thelstnumber{\@arabic\c@lstnumber}
\lst@AddToHook{EveryPar}
    {\global\advance\c@lstnumber\lst@advancelstnum
     \global\advance\c@lstnumber\m@ne \refstepcounter{lstnumber}%
     \lst@SkipOrPrintLabel}%
\global\let\lst@advancelstnum\@ne
```

Note that the counter advances *before* the label is printed and not afterwards. Otherwise we have wrong references—reported by Gregory Van Vooren.

```
\lst@AddToHook{Init}{\def\@currentlabel{\thelstnumber}}
```

The label number is initialized and we ensure correct line numbers for continued listings. An apparently-extraneous advancement of the line number by `-\lst@advancelstnum` when `firstnumber=last` is specified was removed, following a bug report by Joachim Breitner.

```
\lst@AddToHook{InitVars}
    {\global\c@lstnumber\lst@firstnumber
     \global\advance\c@lstnumber\lst@advancenumber
     \global\advance\c@lstnumber-\lst@advancelstnum}
\lst@AddToHook{ExitVars}
    {\global\advance\c@lstnumber\lst@advancelstnum}
```

Walter E. Brown reported problems with pdftex and hyperref. A bad default of `\theHlstlabel` was the reason. Heiko Oberdiek found another bug which was due to the localization of `\lst@neglisting`. He also provided the following fix, replacing `\thelstlisting` with the `\ifx` … `\fi` construction. Ivo Pletikosić reported another problem with the redefinition of `\thelstlisting`. Heiko Oberdiek again provided a fix: `\thelstlisting` must be replaced by `\theHlstlisting`.

```
\AtBeginDocument{%
    \def\theHlstnumber{\ifx\lst@@caption\@empty \lst@neglisting
                                          \else \theHlstlisting \fi
                       .\thelstnumber}}
```

`\lst@skipnumbers` There are more things to do. We calculate how many lines must skip their label. The formula is

`\lst@skipnumbers` = *first printing line* mod `\lst@stepnumber`.

Note that we use a nonpositive representative for `\lst@skipnumbers`.

```
\newcount\lst@skipnumbers % \global
\lst@AddToHook{Init}
    {\ifnum \z@>\lst@stepnumber
         \let\lst@advancelstnum\m@ne
         \edef\lst@stepnumber{-\lst@stepnumber}%
     \fi
     \ifnum \z@<\lst@stepnumber
         \global\lst@skipnumbers\lst@firstnumber
         \global\divide\lst@skipnumbers\lst@stepnumber
         \global\multiply\lst@skipnumbers-\lst@stepnumber
         \global\advance\lst@skipnumbers\lst@firstnumber
         \ifnum\lst@skipnumbers>\z@
             \global\advance\lst@skipnumbers -\lst@stepnumber
         \fi
```

If `\lst@stepnumber` is zero, no line numbers are printed:

```
    \else
        \let\lst@SkipOrPrintLabel\relax
    \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`.

```
\gdef\lst@SkipOrPrintLabel{%
    \ifnum\lst@skipnumbers=\z@
        \global\advance\lst@skipnumbers-\lst@stepnumber\relax
        \lst@PlaceNumber
        \lst@numberfirstlinefalse
    \else
```

If the first line of a listing should get a number, it gets it here.

```
        \lst@ifnumberfirstline
            \lst@PlaceNumber
            \lst@numberfirstlinefalse
        \fi
    \fi
    \global\advance\lst@skipnumbers\@ne}%
```

```
\lst@AddToHook{OnEmptyLine}{%
    \lst@ifnumberblanklines\else \ifnum\lst@skipnumbers=\z@
        \global\advance\lst@skipnumbers-\lst@stepnumber\relax
    \fi\fi}
```

```
\lst@EndAspect
⟨/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.

```
⟨*kernel⟩
\def\lst@parshape{\parshape\@ne \z@ \linewidth}
\lst@AddToHookAtTop{EveryLine}{\lst@parshape}
\lst@AddToHookAtTop{EndGroup}{\lst@parshape}
⟨/kernel⟩
```

Our first aspect in this section.

```
⟨*misc⟩
\lst@BeginAspect{lineshape}
```

`xleftmargin`
`xrightmargin`
`resetmargins`
`linewidth`

Usual stuff.

```
\lst@Key{xleftmargin}{\z@}{\def\lst@xleftmargin{#1}}
\lst@Key{xrightmargin}{\z@}{\def\lst@xrightmargin{#1}}
\lst@Key{resetmargins}{false}[t]{\lstKV@SetIf{#1}\lst@ifresetmargins}
```

The margins become zero if we make an exact box around the listing.

```
\lst@AddToHook{BoxUnsafe}{\let\lst@xleftmargin\z@
                          \let\lst@xrightmargin\z@}
\lst@AddToHook{TextStyle}{%
    \let\lst@xleftmargin\z@ \let\lst@xrightmargin\z@
    \let\lst@ifresetmargins\iftrue}
```

Added above hook after bug report from Magnus Lewis-Smith and José Romildo Malaquias respectively.

```
\lst@Key{linewidth}\linewidth{\def\lst@linewidth{#1}}
\lst@AddToHook{PreInit}{\linewidth\lst@linewidth\relax}
```

`\lst@parshape`

The definition itself is easy.

```
\gdef\lst@parshape{%
    \parshape\@ne \@totalleftmargin \linewidth}
```

We calculate the line width and (inner/outer) indent for a listing.

```
\lst@AddToHook{Init}
    {\lst@ifresetmargins
         \advance\linewidth\@totalleftmargin
         \advance\linewidth\rightmargin
         \@totalleftmargin\z@
     \fi
     \advance\linewidth-\lst@xleftmargin
     \advance\linewidth-\lst@xrightmargin
     \advance\@totalleftmargin\lst@xleftmargin\relax}
```

`lineskip`

The introduction of this key is due to communication with Andreas Bartelt. Version 1.0 implements this feature by redefining `\baselinestretch`.

```
\lst@Key{lineskip}{\z@}{\def\lst@lineskip{#1\relax}}
\lst@AddToHook{Init}
    {\parskip\z@
     \ifdim\z@=\lst@lineskip\else
         \@tempdima\baselineskip
         \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.

```
         \multiply\@tempdima\@cclvi
         \divide\@tempdima\baselineskip\relax
         \multiply\@tempdima\@cclvi
         \edef\baselinestretch{\strip@pt\@tempdima}%
         \selectfont
     \fi}
```

`breaklines`
`breakindent`
`breakautoindent`
`breakatwhitespace`
`prebreak`
`postbreak`

As usual we have no problems in announcing more keys. `breakatwhitespace` is due to Javier Bezos. Unfortunately a previous definition of that key was wrong as Franz Rinnerthaler and Ulrike Fischer reported.

```
\lst@Key{breaklines}{false}[t]{\lstKV@SetIf{#1}\lst@ifbreaklines}
\lst@Key{breakindent}{20pt}{\def\lst@breakindent{#1}}
\lst@Key{breakautoindent}{t}[t]{\lstKV@SetIf{#1}\lst@ifbreakautoindent}
\lst@Key{breakatwhitespace}{false}[t]%
    {\lstKV@SetIf{#1}\lst@ifbreakatwhitespace}
\lst@Key{prebreak}{}{\def\lst@prebreak{#1}}
\lst@Key{postbreak}{}{\def\lst@postbreak{#1}}
```

We assign some different macros and (if necessary) suppress "underfull `\hbox`" messages (and use different pretolerance):

```
\lst@AddToHook{Init}
    {\lst@ifbreaklines
         \hbadness\@M \pretolerance\@M
         \@rightskip\@flushglue \rightskip\@rightskip % \raggedright
         \leftskip\z@skip \parindent\z@
```

A `\raggedright` above has been replaced by setting the values by hand after a bug report from Morten Høgholm.

We use the normal parshape and the calculated `\lst@breakshape` (see below).

```
         \def\lst@parshape{\parshape\tw@ \@totalleftmargin\linewidth
                           \lst@breakshape}%
      \else
         \let\lst@discretionary\@empty
      \fi}
\lst@AddToHook{OnNewLine}
    {\lst@ifbreaklines \lst@breakNewLine \fi}
```

`\lst@discretionary` `\lst@spacekern` Here comes the whole magic: We set a discretionary break after each 'output unit'. However we redefine `\space` to be used inside `\discretionary` and use `EveryLine` hook. After a bug report by Carsten Hamm I've added `\kern-\lst@xleftmargin`, which became `\kern-\@totalleftmargin` after a bug report by Christian Kaiser.

```
\gdef\lst@discretionary{%
    \lst@ifbreakatwhitespace
        \lst@ifwhitespace \lst@@discretionary \fi
    \else
        \lst@@discretionary
    \fi}%
\gdef\lst@@discretionary{%
    \discretionary{\let\space\lst@spacekern\lst@prebreak}%
                 {\llap{\lsthk@EveryLine
                   \kern\lst@breakcurrindent \kern-\@totalleftmargin}%
                   \let\space\lst@spacekern\lst@postbreak}{}}
\lst@AddToHook{PostOutput}{\lst@discretionary}
\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 indention (coming from white spaces at the beginning of the line) if 'auto indent' is on.

```
\gdef\lst@breakNewLine{%
    \@tempdima\lst@breakindent\relax
    \lst@ifbreakautoindent \advance\@tempdima\lst@lostspace \fi
```

Now we calculate the margin and line width of the wrapped part ...

```
    \@tempdimc-\@tempdima \advance\@tempdimc\linewidth
                          \advance\@tempdima\@totalleftmargin
```

... and store it in `\lst@breakshape`.

```
    \xdef\lst@breakshape{\noexpand\lst@breakcurrindent \the\@tempdimc}%
    \xdef\lst@breakcurrindent{\the\@tempdima}}
\global\let\lst@breakcurrindent\z@ % init
```

The initialization of `\lst@breakcurrindent` has been added after a bug report by Alvaro Herrera.

To do: We could speed this up by allocating two global dimensions.

`\lst@breakshape` Andreas Deininger reported a problem which is resolved by providing a default break shape.

```
\gdef\lst@breakshape{\@totalleftmargin \linewidth}
```

`\lst@breakProcessOther` is the same as `\lst@ProcessOther` except that it also outputs the current token string. This inserts a potential linebreak point. Only the closing parenthesis uses this macro yet.

```
\gdef\lst@breakProcessOther#1{\lst@ProcessOther#1\lst@OutputOther}
\lst@AddToHook{SelectCharTable}
    {\lst@ifbreaklines \lst@Def{`)}{\lst@breakProcessOther)}\fi}
```

A bug reported by Gabriel Tauro has been removed by using `\lst@ProcessOther` instead of `\lst@AppendOther`.

```
\lst@EndAspect
⟨/misc⟩
```

17.5 Frames

Another aspect.

```
⟨*misc⟩
\lst@BeginAspect[lineshape]{frames}
```

`framexleftmargin` `framexrightmargin` `framextopmargin` `framexbottommargin` These keys just save the argument.

```
\lst@Key{framexleftmargin}{\z@}{\def\lst@framexleftmargin{#1}}
\lst@Key{framexrightmargin}{\z@}{\def\lst@framexrightmargin{#1}}
\lst@Key{framextopmargin}{\z@}{\def\lst@framextopmargin{#1}}
\lst@Key{framexbottommargin}{\z@}{\def\lst@framexbottommargin{#1}}
```

`backgroundcolor` Ralf Imhäuser inspired the key `backgroundcolor`. All keys save the argument, and ...

```
\lst@Key{backgroundcolor}{}{\def\lst@bkgcolor{#1}}
\lst@Key{fillcolor}{}{\def\lst@fillcolor{#1}}
\lst@Key{rulecolor}{}{\def\lst@rulecolor{#1}}
\lst@Key{rulesepcolor}{}{\def\lst@rulesepcolor{#1}}
```

... some have default settings if they are empty.

```
\lst@AddToHook{Init}{%
    \ifx\lst@fillcolor\@empty
        \let\lst@fillcolor\lst@bkgcolor
    \fi
    \ifx\lst@rulesepcolor\@empty
        \let\lst@rulesepcolor\lst@fillcolor
    \fi}
```

`rulesep` `framerule` `framesep` `frameshape` Another set of keys, which mainly save their respective argument. `frameshape` capitalizes all letters, and checks whether at least one round corner is specified. Eventually we define `\lst@frame` to be empty if and only if there is no frameshape.

```
\lst@Key{rulesep}{2pt}{\def\lst@rulesep{#1}}
```

```
\lst@Key{framerule}{.4pt}{\def\lst@framerulewidth{#1}}
\lst@Key{framesep}{3pt}{\def\lst@frametextsep{#1}}
\lst@Key{frameshape}{}{%
    \let\lst@xrulecolor\@empty
    \lstKV@FourArg{#1}%
    {\uppercase{\def\lst@frametshape{##1}}%
     \uppercase{\def\lst@framelshape{##2}}%
     \uppercase{\def\lst@framershape{##3}}%
     \uppercase{\def\lst@framebshape{##4}}%
     \let\lst@ifframeround\iffalse
     \lst@IfSubstring R\lst@frametshape{\let\lst@ifframeround\iftrue}{}%
     \lst@IfSubstring R\lst@framebshape{\let\lst@ifframeround\iftrue}{}%
     \def\lst@frame{##1##2##3##4}}}
```

frameround
frame

We have to do some conversion here.

```
\lst@Key{frameround}\relax
    {\uppercase{\def\lst@frameround{#1}}%
     \expandafter\lstframe@\lst@frameround ffff\relax}
\global\let\lst@frameround\@empty
```

In case of an verbose argument, we use the `trbl`-subset replacement.

```
\lst@Key{frame}\relax{%
    \let\lst@xrulecolor\@empty
    \lstKV@SwitchCases{#1}%
    {none:\let\lst@frame\@empty\\%
     leftline:\def\lst@frame{l}\\%
     topline:\def\lst@frame{t}\\%
     bottomline:\def\lst@frame{b}\\%
     lines:\def\lst@frame{tb}\\%
     single:\def\lst@frame{trbl}\\%
     shadowbox:\def\lst@frame{tRBl}%
            \def\lst@xrulecolor{\lst@rulesepcolor}%
            \def\lst@rulesep{\lst@frametextsep}%
    }{\def\lst@frame{#1}}%
    \expandafter\lstframe@\lst@frameround ffff\relax}
```

Adding `t`, `r`, `b`, and `l` in case of their upper case versions makes later tests easier.

```
\gdef\lstframe@#1#2#3#4#5\relax{%
    \lst@IfSubstring T\lst@frame{\edef\lst@frame{t\lst@frame}}{}%
    \lst@IfSubstring R\lst@frame{\edef\lst@frame{r\lst@frame}}{}%
    \lst@IfSubstring B\lst@frame{\edef\lst@frame{b\lst@frame}}{}%
    \lst@IfSubstring L\lst@frame{\edef\lst@frame{l\lst@frame}}{}%
```

We now check top and bottom frame rules, …

```
    \let\lst@frametshape\@empty \let\lst@framebshape\@empty
    \lst@frameCheck
        ltr\lst@framelshape\lst@frametshape\lst@framershape #4#1%
    \lst@frameCheck
        LTR\lst@framelshape\lst@frametshape\lst@framershape #4#1%
    \lst@frameCheck
        lbr\lst@framelshape\lst@framebshape\lst@framershape #3#2%
    \lst@frameCheck
        LBR\lst@framelshape\lst@framebshape\lst@framershape #3#2%
```

… look for round corners …

```
    \let\lst@ifframeround\iffalse
    \lst@IfSubstring R\lst@frametshape{\let\lst@ifframeround\iftrue}{}%
    \lst@IfSubstring R\lst@framebshape{\let\lst@ifframeround\iftrue}{}%
```

and define left and right frame shape.

```
    \let\lst@framelshape\@empty \let\lst@framershape\@empty
    \lst@IfSubstring L\lst@frame
        {\def\lst@framelshape{YY}}%
        {\lst@IfSubstring l\lst@frame{\def\lst@framelshape{Y}}{}}%
    \lst@IfSubstring R\lst@frame
        {\def\lst@framershape{YY}}%
        {\lst@IfSubstring r\lst@frame{\def\lst@framershape{Y}}{}}}
```

Now comes the macro used to define top and bottom frame shape. It extends the macro `#5`. The last two arguments show whether left and right corners are round. `#4` and `#6` are temporary macros. `#1#2#3` are the three characters we test for.

```
\gdef\lst@frameCheck#1#2#3#4#5#6#7#8{%
    \lst@IfSubstring #1\lst@frame
        {\if #7T\def#4{R}\else \def#4{Y}\fi}%
        {\def#4{N}}%
    \lst@IfSubstring #3\lst@frame
        {\if #8T\def#6{R}\else \def#6{Y}\fi}%
        {\def#6{N}}%
    \lst@IfSubstring #2\lst@frame{\edef#5{#5#4Y#6}}{}}
```

For text style listings all frames and the background color are deactivated – added after bug reports by Stephen Reindl and Thomas ten Cate

```
\lst@AddToHook{TextStyle}
   {\let\lst@frame\@empty
    \let\lst@frametshape\@empty
    \let\lst@framershape\@empty
    \let\lst@framebshape\@empty
    \let\lst@framelshape\@empty
    \let\lst@bkgcolor\@empty}
```

As per a bug report by Ignacio Fernández Galván, the small section of background color to the left of the margin is now drawn before the left side of the frame is drawn, so that they overlap correctly in Acrobat.

`\lst@frameMakeVBox`

```
\gdef\lst@frameMakeBoxV#1#2#3{%
    \setbox#1\hbox{%
      \color@begingroup \lst@rulecolor
      \ifx\lst@framelshape\@empty
      \else
            \llap{%
                \lst@frameBlock\lst@fillcolor\lst@frametextsep{#2}{#3}%
                \kern\lst@framexleftmargin}%
      \fi
      \llap{\setbox\z@\hbox{\vrule\@width\z@\@height#2\@depth#3%
                            \lst@frameL}%
            \rlap{\lst@frameBlock\lst@rulesepcolor{\wd\z@}%
                                                  {\ht\z@}{\dp\z@}}%
            \box\z@
            \kern\lst@frametextsep\relax
            \kern\lst@framexleftmargin}%
      \rlap{\kern-\lst@framexleftmargin
                    \@tempdima\linewidth
            \advance\@tempdima\lst@framexleftmargin
            \advance\@tempdima\lst@framexrightmargin
            \lst@frameBlock\lst@bkgcolor\@tempdima{#2}{#3}%
            \ifx\lst@framershape\@empty
                \kern\lst@frametextsep\relax
            \else
                \lst@frameBlock\lst@fillcolor\lst@frametextsep{#2}{#3}%
            \fi
            \setbox\z@\hbox{\vrule\@width\z@\@height#2\@depth#3%
                            \lst@frameR}%
            \rlap{\lst@frameBlock\lst@rulesepcolor{\wd\z@}%
                                                  {\ht\z@}{\dp\z@}}%
            \box\z@}%
      \color@endgroup}}
```

`\lst@frameBlock`

```
\gdef\lst@frameBlock#1#2#3#4{%
    \color@begingroup
      #1%
      \setbox\z@\hbox{\vrule\@height#3\@depth#4%
                      \ifx#1\@empty \@width\z@ \kern#2\relax
                              \else \@width#2\relax \fi}%
      \box\z@
    \color@endgroup}
```

`\lst@frameR` typesets right rules. We only need to iterate through `\lst@framershape`.

```
\gdef\lst@frameR{%
    \expandafter\lst@frameR@\lst@framershape\relax
    \kern-\lst@rulesep}
\gdef\lst@frameR@#1{%
    \ifx\relax#1\@empty\else
        \if #1Y\lst@framevrule \else \kern\lst@framerulewidth \fi
        \kern\lst@rulesep
        \expandafter\lst@frameR@b
    \fi}
\gdef\lst@frameR@b#1{%
    \ifx\relax#1\@empty
    \else
        \if #1Y\color@begingroup
               \lst@xrulecolor
               \lst@framevrule
               \color@endgroup
        \else
               \kern\lst@framerulewidth
        \fi
        \kern\lst@rulesep
        \expandafter\lst@frameR@
    \fi}
```

`\lst@frameL` Ditto left rules.

```
\gdef\lst@frameL{%
    \kern-\lst@rulesep
    \expandafter\lst@frameL@\lst@framelshape\relax}
\gdef\lst@frameL@#1{%
    \ifx\relax#1\@empty\else
        \kern\lst@rulesep
        \if#1Y\lst@framevrule \else \kern\lst@framerulewidth \fi
        \expandafter\lst@frameL@
    \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.

```
\gdef\lst@frameH#1#2{%
    \global\let\lst@framediml\z@ \global\let\lst@framedimr\z@
    \setbox\z@\hbox{}\@tempcntb\z@
    \expandafter\lst@frameH@\expandafter#1#2\relax\relax\relax
            \@tempdimb\lst@frametextsep\relax
    \advance\@tempdimb\lst@framerulewidth\relax
            \@tempdimc-\@tempdimb
    \advance\@tempdimc\ht\z@
    \advance\@tempdimc\dp\z@
    \setbox\z@=\hbox{%
      \lst@frameHBkg\lst@fillcolor\@tempdimb\@firstoftwo
      \if#1T\rlap{\raise\dp\@tempboxa\box\@tempboxa}%
       \else\rlap{\lower\ht\@tempboxa\box\@tempboxa}\fi
      \lst@frameHBkg\lst@rulesepcolor\@tempdimc\@secondoftwo
      \advance\@tempdimb\ht\@tempboxa
      \if#1T\rlap{\raise\lst@frametextsep\box\@tempboxa}%
       \else\rlap{\lower\@tempdimb\box\@tempboxa}\fi
      \rlap{\box\z@}%
    }}
\gdef\lst@frameH@#1#2#3#4{%
    \ifx\relax#4\@empty\else
        \lst@frameh \@tempcntb#1#2#3#4%
        \advance\@tempcntb\@ne
        \expandafter\lst@frameH@\expandafter#1%
    \fi}
\gdef\lst@frameHBkg#1#2#3{%
    \setbox\@tempboxa\hbox{%
        \kern-\lst@framexleftmargin
        #3{\kern-\lst@framediml\relax}{\@tempdima\z@}%
        \ifdim\lst@framediml>\@tempdimb
            #3{\@tempdima\lst@framediml \advance\@tempdima-\@tempdimb
               \lst@frameBlock\lst@rulesepcolor\@tempdima\@tempdimb\z@}%
              {\kern-\lst@framediml
               \advance\@tempdima\lst@framediml\relax}%
        \fi
        #3{\@tempdima\z@
           \ifx\lst@framelshape\@empty\else
               \advance\@tempdima\@tempdimb
           \fi
           \ifx\lst@framershape\@empty\else
               \advance\@tempdima\@tempdimb
           \fi}%
          {\ifdim\lst@framedimr>\@tempdimb
```

```
              \advance\@tempdima\lst@framedimr\relax
           \fi}%
        \advance\@tempdima\linewidth
        \advance\@tempdima\lst@framexleftmargin
        \advance\@tempdima\lst@framexrightmargin
        \lst@frameBlock#1\@tempdima#2\z@
        #3{\ifdim\lst@framedimr>\@tempdimb
               \@tempdima-\@tempdimb
               \advance\@tempdima\lst@framedimr\relax
               \lst@frameBlock\lst@rulesepcolor\@tempdima\@tempdimb\z@
           \fi}{}%
        }}
```

`\lst@frameh` This is the low-level macro used to draw top and bottom frame rules. It *adds* one rule plus corners to box 0. The first parameter gives the size of the corners and the second is either `T` or `B`. `#3#4#5` is a left-to-right description of the frame and is in {`Y,N,R`} × {`Y,N`} × {`Y,N,R`}. We move to the correct horizontal position, set the left corner, the horizontal line, and the right corner.

```
\gdef\lst@frameh#1#2#3#4#5{%
   \lst@frameCalcDimA#1%
   \lst@ifframeround \@getcirc\@tempdima \fi

    \setbox\z@\hbox{%
      \begingroup
      \setbox\z@\hbox{%
        \kern-\lst@framexleftmargin
        \color@begingroup
        \ifnum#1=\z@ \lst@rulecolor \else \lst@xrulecolor \fi
```

`\lst@frameCorner` gets four arguments: `\llap`, `TL` or `BL`, the corner type ∈ {`Y,N,R`}, and the size `#1`.

```
        \lst@frameCornerX\llap{#2L}#3#1%
        \ifdim\lst@framediml<\@tempdimb
            \xdef\lst@framediml{\the\@tempdimb}%
        \fi
        \begingroup
        \if#4Y\else \let\lst@framerulewidth\z@ \fi
                \@tempdima\lst@framexleftmargin
        \advance\@tempdima\lst@framexrightmargin
        \advance\@tempdima\linewidth
        \vrule\@width\@tempdima\@height\lst@framerulewidth \@depth\z@
        \endgroup
        \lst@frameCornerX\rlap{#2R}#5#1%
        \ifdim\lst@framedimr<\@tempdimb
            \xdef\lst@framedimr{\the\@tempdimb}%
        \fi
        \color@endgroup}%

      \if#2T\rlap{\raise\dp\z@\box\z@}%
       \else\rlap{\lower\ht\z@\box\z@}\fi
      \endgroup
      \box\z@}}
```

`\lst@frameCornerX` typesets a single corner and returns `\@tempdimb`, the width of the corner.

```
\gdef\lst@frameCornerX#1#2#3#4{%
```

```
    \setbox\@tempboxa\hbox{\csname\@lst @frame\if#3RR\fi #2\endcsname}%
    \@tempdimb\wd\@tempboxa
    \if #3R%
        #1{\box\@tempboxa}%
    \else
        \if #3Y\expandafter#1\else
               \@tempdimb\z@ \expandafter\vphantom \fi
        {\box\@tempboxa}%
    \fi}
```

`\lst@frameCalcDimA` calculates an all over width; used by `\lst@frameh` and `\lst@frameInit`.

```
\gdef\lst@frameCalcDimA#1{%
            \@tempdima\lst@rulesep
    \advance\@tempdima\lst@framerulewidth
    \multiply\@tempdima#1\relax
    \advance\@tempdima\lst@frametextsep
    \advance\@tempdima\lst@framerulewidth
    \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.

```
\lst@AddToHook{Init}{\lst@frameInit}
\newbox\lst@framebox
\gdef\lst@frameInit{%
    \ifx\lst@framelshape\@empty \let\lst@frameL\@empty \fi
    \ifx\lst@framershape\@empty \let\lst@frameR\@empty \fi
    \def\lst@framevrule{\vrule\@width\lst@framerulewidth\relax}%
```

We adjust values to round corners if necessary.

```
    \lst@ifframeround
        \lst@frameCalcDimA\z@ \@getcirc\@tempdima
        \@tempdimb\@tempdima \divide\@tempdimb\tw@
        \advance\@tempdimb -\@wholewidth
        \edef\lst@frametextsep{\the\@tempdimb}%
        \edef\lst@framerulewidth{\the\@wholewidth}%

        \lst@frameCalcDimA\@ne \@getcirc\@tempdima
        \@tempdimb\@tempdima \divide\@tempdimb\tw@
        \advance\@tempdimb -\tw@\@wholewidth
        \advance\@tempdimb -\lst@frametextsep
        \edef\lst@rulesep{\the\@tempdimb}%
    \fi

    \lst@frameMakeBoxV\lst@framebox{\ht\strutbox}{\dp\strutbox}%
    \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`?

```
    \ifx\lst@frametshape\@empty\else
        \lst@frameH T\lst@frametshape
        \ifvoid\z@\else
            \par\lst@parshape
            \@tempdima-\baselineskip \advance\@tempdima\ht\z@
            \ifdim\prevdepth<\@cclvi\p@\else
```

```
                \advance\@tempdima\prevdepth
            \fi
            \ifdim\@tempdima<\z@
                \vskip\@tempdima\vskip\lineskip
            \fi
            \noindent\box\z@\par
            \lineskiplimit\maxdimen \lineskip\z@
        \fi
        \lst@frameSpreadV\lst@framextopmargin
    \fi}
```

`\parshape\lst@parshape` ensures that the top rules correctly indented. The bug was reported by Marcin Kasperski.

We typeset left and right rules every line.

```
\lst@AddToHook{EveryLine}{\lst@framelr}
\global\let\lst@framelr\@empty
```

`\lst@frameExit` The rules at the bottom.

```
\lst@AddToHook{DeInit}
    {\ifx\lst@framebshape\@empty\else \lst@frameExit \fi}
\gdef\lst@frameExit{%
    \lst@frameSpreadV\lst@framexbottommargin
    \lst@frameH B\lst@framebshape
    \ifvoid\z@\else
        \everypar{}\par\lst@parshape\nointerlineskip\noindent\box\z@
    \fi}
```

`\lst@frameSpreadV` sets rules for vertical spread.

```
\gdef\lst@frameSpreadV#1{%
    \ifdim\z@=#1\else
        \everypar{}\par\lst@parshape\nointerlineskip\noindent
        \lst@frameMakeBoxV\z@{#1}{\z@}%
        \box\z@
    \fi}
```

`\lst@frameTR`
`\lst@frameBR`
`\lst@frameBL`
`\lst@frameTL`

These macros make a vertical and horizontal rule. The implicit argument `\@tempdima` gives the size of two corners and is provided by `\lst@frameh`.

```
\gdef\lst@frameTR{%
    \vrule\@width.5\@tempdima\@height\lst@framerulewidth\@depth\z@
    \kern-\lst@framerulewidth
    \raise\lst@framerulewidth\hbox{%
        \vrule\@width\lst@framerulewidth\@height\z@\@depth.5\@tempdima}}
\gdef\lst@frameBR{%
    \vrule\@width.5\@tempdima\@height\lst@framerulewidth\@depth\z@
    \kern-\lst@framerulewidth
    \vrule\@width\lst@framerulewidth\@height.5\@tempdima\@depth\z@}
\gdef\lst@frameBL{%
    \vrule\@width\lst@framerulewidth\@height.5\@tempdima\@depth\z@
    \kern-\lst@framerulewidth
    \vrule\@width.5\@tempdima\@height\lst@framerulewidth\@depth\z@}
\gdef\lst@frameTL{%
    \raise\lst@framerulewidth\hbox{%
        \vrule\@width\lst@framerulewidth\@height\z@\@depth.5\@tempdima}%
    \kern-\lst@framerulewidth
    \vrule\@width.5\@tempdima\@height\lst@framerulewidth\@depth\z@}
```

`\lst@frameRoundT`
`\lst@frameRoundB`

are helper macros to typeset round corners. We set height and depth to the visible parts of the circle font.

```
\gdef\lst@frameRoundT{%
    \setbox\@tempboxa\hbox{\@circlefnt\char\@tempcnta}%
    \ht\@tempboxa\lst@framerulewidth
    \box\@tempboxa}
\gdef\lst@frameRoundB{%
    \setbox\@tempboxa\hbox{\@circlefnt\char\@tempcnta}%
    \dp\@tempboxa\z@
    \box\@tempboxa}
```

`\lst@frameRTR`
`\lst@frameRBR`
`\lst@frameRBL`
`\lst@frameRTL`

The round corners.

```
\gdef\lst@frameRTR{%
    \hb@xt@.5\@tempdima{\kern-\lst@framerulewidth
                           \kern.5\@tempdima \lst@frameRoundT \hss}}
\gdef\lst@frameRBR{%
    \hb@xt@.5\@tempdima{\kern-\lst@framerulewidth
    \advance\@tempcnta\@ne \kern.5\@tempdima \lst@frameRoundB \hss}}
\gdef\lst@frameRBL{%
    \advance\@tempcnta\tw@ \lst@frameRoundB
    \kern-.5\@tempdima}
\gdef\lst@frameRTL{%
    \advance\@tempcnta\thr@@\lst@frameRoundT
    \kern-.5\@tempdima}
```

```
\lst@EndAspect
⟨/misc⟩
```

17.6 Macro use for make

`\lst@makemode`
`\lst@ifmakekey`

If we've entered the special mode for Make, we save whether the last identifier has been a first order keyword.

```
⟨*misc⟩
\lst@BeginAspect[keywords]{make}
```

```
\lst@NewMode\lst@makemode
\lst@AddToHook{Output}{%
    \ifnum\lst@mode=\lst@makemode
        \ifx\lst@thestyle\lst@gkeywords@sty
            \lst@makekeytrue
        \fi
    \fi}
```

```
\gdef\lst@makekeytrue{\let\lst@ifmakekey\iftrue}
\gdef\lst@makekeyfalse{\let\lst@ifmakekey\iffalse}
\global\lst@makekeyfalse % init
```

`makemacrouse`

adjusts the character table if necessary

```
\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.

```
\gdef\lst@MakeSCT{%
    \lst@ifmakemacrouse
        \lst@ReplaceInput{$(}{%
            \lst@PrintToken
            \lst@EnterMode\lst@makemode{\lst@makekeyfalse}%
            \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 `)`).

```
        \lst@ReplaceInput{)}{%
            \ifnum\lst@mode=\lst@makemode
                \lst@PrintToken
                \begingroup
                    \lst@ProcessOther)%
                    \lst@ifmakekey
                        \let\lst@currstyle\lst@gkeywords@sty
                    \fi
                    \lst@OutputOther
                \endgroup
                \lst@LeaveMode
            \else
                \expandafter\lst@ProcessOther\expandafter)%
            \fi}%
```

If `makemacrouse` is off then both `$(` are just ‘others’.

```
    \else
        \lst@ReplaceInput{$(}{\lst@ProcessOther\$\lst@ProcessOther(}%
    \fi}

\lst@EndAspect
⟨/misc⟩
```

18 Typesetting a listing

```
⟨*kernel⟩
```

`\lst@lineno`
`print`
`firstline`
`lastline`

The ‘current line’ counter and three option keys.

```
\newcount\lst@lineno % \global
\lst@AddToHook{InitVars}{\global\lst@lineno\@ne}

\lst@Key{print}{true}[t]{\lstKV@SetIf{#1}\lst@ifprint}
\lst@Key{firstline}\relax{\def\lst@firstline{#1\relax}}
\lst@Key{lastline}\relax{\def\lst@lastline{#1\relax}}
```

Initialize the linerange with reasonable values

```
\lst@AddToHook{PreSet}
{%
  \def\lst@firstline{1\relax}
  \def\lst@lastline{9999999\relax}
  \let\lst@linerange\@empty
}
```

The following code is just copied from the current development version, and from the `lstpatch.sty` file that Carsten left in version 1.3b for doing line ranges with numbers and range markers. It uses some more keys:

`linerange`
`consecutivenumbers`
`rangeprefix`
`rangesuffix`
`rangebeginprefix`
`rangebeginsuffix`
`rangeendprefix`
`rangeendsuffix`
`includerangemarker`

First, the options that control the line-range handling.

```
\lst@Key{linerange}\relax{\lstKV@OptArg[]{#1}{%
    \def\lst@interrange{##1}\def\lst@linerange{##2,}}}
\lst@Key{consecutivenumbers}{true}[t]{%
  \lstKV@SetIf{#1}\lst@ifconsecutivenumbers}
```

The next options are needed for an easy description of arbitrary linerange markers:

```
\lst@Key{rangeprefix}\relax{\def\lst@rangebeginprefix{#1}%
                            \def\lst@rangeendprefix{#1}}
\lst@Key{rangesuffix}\relax{\def\lst@rangebeginsuffix{#1}%
                            \def\lst@rangeendsuffix{#1}}
\lst@Key{rangebeginprefix}{}{\def\lst@rangebeginprefix{#1}}
\lst@Key{rangebeginsuffix}{}{\def\lst@rangebeginsuffix{#1}}
\lst@Key{rangeendprefix}{}{\def\lst@rangeendprefix{#1}}
\lst@Key{rangeendsuffix}{}{\def\lst@rangeendsuffix{#1}}
\lst@Key{includerangemarker}{true}[t]{\lstKV@SetIf{#1}%
  \lst@ifincluderangemarker}
```

The line range is known—it is set either explicitly (by numbers or by arbitrary linerange markers) or implicitly from the default values (1 and 999999). `\lst@GetLineInterval` parses the known line range recursively, because there may be a comma separated list of pairs.

```
\lst@AddToHook{Init}
{%
  \ifx\lst@linerange\@empty
      \edef\lst@linerange{{\lst@firstline}-{\lst@lastline},}%
  \fi%
  \lst@GetLineInterval%
}%
\def\lst@GetLineInterval{\expandafter\lst@GLI\lst@linerange\@nil}
```

Splitting the comma separated list of pairs is done by

```
\def\lst@GLI#1,#2\@nil{%
```

The list is split into two parts: the first element (#1) of the list and the rest of the list (#2) which is stored in `\lst@linerange` to be processed later.

```
  \def\lst@linerange{#2}\lst@GLI@#1--\@nil%
}
```

Further parsing is done by `lst@GLI@`, the argument is `#1--\@nil` that is the first element of the list augmented by a sentinel. This argument is parsed with the template `#1-#2-#3\@nil`. A linerange given as e. g. '`12-21`' leads to the argument `12-21--\@nil`, so it is parsed as #1← `12`, #2←`21`, and #3←`-`.

```
\def\lst@GLI@#1-#2-#3\@nil{%
```

First we test, if the linerange starts with a number (consisting of `-`, 1, 2, 3, . . .) or an arbitrary linerange marker. If a linerange starts erroneously with a '`-`' character, the argument #1 isn't set and so the range starts with 1, so giving a perhaps expected result, but the definition of a linerange must always consist at least of the three parts ⟨*first*⟩, '`-`', and ⟨*last*⟩.

```
  \lst@IfNumber{#1}%
  {%
    \ifx\@empty#1\@empty
        \let\lst@firstline\@ne
    \else
```

```
        \def\lst@firstline{#1\relax}%
    \fi
```

Now we know the starting number of the linerange. #3 isn't set with a linerange consisting of a single number, in all other cases #2 defines the last line (explicitly or implicitly).

```
    \ifx\@empty#3\@empty%
        \def\lst@lastline{9999999\relax}%
        \ifx\@empty#2\@empty
            \let\lst@lastline\lst@firstline%
        \fi
    \else%
        \ifx\@empty#2\@empty
            \def\lst@lastline{9999999\relax}
        \else % doesn't happen(?)
            \def\lst@lastline{#2\relax}%
        \fi
    \fi
  }%
```

If we've found an arbitrary linerange marker, we set firstline and lastline to 9999999. This prevents (almost) anything from being printed for now.

```
  {%
    \def\lst@firstline{9999999\relax}%
    \let\lst@lastline\lst@firstline
```

We add the prefixes and suffixes to the markers.

```
    \let\lst@rangebegin\lst@rangebeginprefix
    \lst@AddTo\lst@rangebegin{#1}%
    \lst@Extend\lst@rangebegin\lst@rangebeginsuffix
    \ifx\@empty#3\@empty
        \let\lst@rangeend\lst@rangeendprefix
        \lst@AddTo\lst@rangeend{#1}
        \lst@Extend\lst@rangeend\lst@rangeendsuffix
    \else
        \ifx\@empty#2\@empty
            \let\lst@rangeend\@empty
        \else
            \let\lst@rangeend\lst@rangeendprefix
            \lst@AddTo\lst@rangeend{#2}%
            \lst@Extend\lst@rangeend\lst@rangeendsuffix
        \fi
    \fi
```

The following definition will be executed in the `SelectCharTable` hook and here right now if we are already processing a listing.

```
    \global\def\lst@DefRange{%
      \expandafter\lst@CArgX\lst@rangebegin\relax\lst@DefRangeB}%
    \ifnum\lst@mode=\lst@Pmode \expandafter\lst@DefRange \fi%
  }%
}
```

`\lst@DefRange` `\lst@DefRange` is not inserted via a hook anymore. Instead it is now called directly from `\lst@SelectCharTable`. This was necessary to get rid of an interference with the escape-to-LaTeX-feature. The bug was reported by Michael Bachmann.

Another change is due to the same bug: `\lst@DefRange` is redefined globally when the begin of code is found, see below. The bug was reported by Tobias Rapp and Markus Luisser.

```
\lst@AddToHookExe{DeInit}{\global\let\lst@DefRange\@empty}
```

Actually defining the marker (via `\lst@GLI@`, `\lst@DefRange`, `\lst@CArgX` as seen above) is similar to `\lst@DefDelimB`—except that we unfold the first parameter and use different ⟨*execute*⟩, ⟨*pre*⟩, and ⟨*post*⟩ statements.

```
\def\lst@DefRangeB#1#2{\lst@DefRangeB@#1#2}
\def\lst@DefRangeB@#1#2#3#4{%
    \lst@CDef{#1{#2}{#3}}#4{}%
    {\lst@ifincluderangemarker
         \lst@LeaveMode
         \let#1#4%
         \lst@DefRangeEnd
         \lst@InitLstNumber
     \else
         \@tempcnta\lst@lineno \advance\@tempcnta\@ne
         \edef\lst@firstline{\the\@tempcnta\relax}%
         \gdef\lst@OnceAtEOL{\let#1#4\lst@DefRangeEnd}%
         \lst@InitLstNumber
     \fi
 \global\let\lst@DefRange\lst@DefRangeEnd
     \lst@CArgEmpty}%
    \@empty}
```

Modify labels and define `\lst@InitLstNumber` used above according to an error reported by Omair-Inam Abdul-Matin.

```
\def\lstpatch@labels{%
\gdef\lst@SetFirstNumber{%
    \ifx\lst@firstnumber\@undefined
        \@tempcnta 0\csname\@lst no@\lst@intname\endcsname\relax
        \ifnum\@tempcnta=\z@ \else
            \lst@nololtrue
            \advance\@tempcnta\lst@advancenumber
            \edef\lst@firstnumber{\the\@tempcnta\relax}%
        \fi
    \fi}%
}
\def\lst@InitLstNumber{%
     \global\c@lstnumber\lst@firstnumber
     \global\advance\c@lstnumber\lst@advancenumber
     \global\advance\c@lstnumber-\lst@advancelstnum
     \ifx \lst@firstnumber\c@lstnumber
         \global\advance\c@lstnumber-\lst@advancelstnum
     \fi%
```

Byron K. Boulton reported, that the line numbers are off by one, if they are displayed when a linerange is given by patterns and `includerangemarker=false` is set. Adding this test corrects this behaviour.

```
     \lst@ifincluderangemarker\else%
         \global\advance\c@lstnumber by 1%
     \fi%
     }
```

The end-marker is defined if and only if it's not empty. The definition is similar to `\lst@DefDelimE`—with the above exceptions and except that we define the re-entry point `\lst@DefRangeE@@` as it is defined in the new version of `\lst@MProcessListing` above.

```
\def\lst@DefRangeEnd{%
    \ifx\lst@rangeend\@empty\else
        \expandafter\lst@CArgX\lst@rangeend\relax\lst@DefRangeE
    \fi}
\def\lst@DefRangeE#1#2{\lst@DefRangeE@#1#2}
\def\lst@DefRangeE@#1#2#3#4{%
    \lst@CDef{#1#2{#3}}#4{}%
    {\let#1#4%
     \edef\lst@lastline{\the\lst@lineno\relax}%
     \lst@DefRangeE@@}%
    \@empty}
\def\lst@DefRangeE@@#1\@empty{%
    \lst@ifincluderangemarker
        #1\lst@XPrintToken
    \fi
    \lst@LeaveModeToPmode
    \lst@BeginDropInput{\lst@Pmode}}
\def\lst@LeaveModeToPmode{%
    \ifnum\lst@mode=\lst@Pmode
        \expandafter\lsthk@EndGroup
    \else
        \expandafter\egroup\expandafter\lst@LeaveModeToPmode
    \fi}
```

Sometimes it is good to have two or more excerpts of one program numbered consecutively, i. e. 1–11 instead of 3–5, 12–17, 20–21. So we introduce `\lst@DisplayConsecutiveNumbersOrNot`, which corrects the displayed line numbers to be always equal to the input line numbers

```
\def\lst@DisplayConsecutiveNumbersOrNot{%
    \lst@ifconsecutivenumbers\else%
        \c@lstnumber=\numexpr-1+\lst@lineno
    \fi%
}
```

Eventually we shouldn't forget to install `\lst@OnceAtEOL`, which must also be called in `\lst@MSkipToFirst`.

```
\lst@AddToHook{EOL}{\lst@OnceAtEOL\global\let\lst@OnceAtEOL\@empty}
\gdef\lst@OnceAtEOL{}% Init
```

The following code was introduced in listings version 1.4. The code resembles the lines 14031–14061 of version 1.3c, but contains loops:
`\def\lst@next{...}\expandafter\lst@next`.
In fact that code will never be expanded, because there are two analoguous definitions in section 18.5 below, so they override these two commands.

```
\def\lst@MSkipToFirst{%
    \global\advance\lst@lineno\@ne
    \ifnum \lst@lineno=\lst@firstline
        \def\lst@next{\lst@LeaveMode \global\lst@newlines\z@
        \lst@OnceAtEOL \global\let\lst@OnceAtEOL\@empty
        \lst@InitLstNumber % Added to work with modified \lsthk@PreInit.
        \lsthk@InitVarsBOL
        \lst@BOLGobble}%
        \expandafter\lst@next
    \fi}
\def\lst@SkipToFirst{%
    \ifnum \lst@lineno<\lst@firstline
        \def\lst@next{\lst@BeginDropInput\lst@Pmode
        \lst@Let{13}\lst@MSkipToFirst
        \lst@Let{10}\lst@MSkipToFirst}%
        \expandafter\lst@next
    \else
        \expandafter\lst@BOLGobble
    \fi}
```

Finally the service macro `\lst@IfNumber`:

```
\def\lst@IfNumber#1{%
    \ifx\@empty#1\@empty
        \let\lst@next\@firstoftwo
    \else
        \lst@IfNumber@#1\@nil
    \fi
    \lst@next}
\def\lst@IfNumber@#1#2\@nil{%
    \let\lst@next\@secondoftwo
    \ifnum`#1>47\relax \ifnum`#1>57\relax\else
        \let\lst@next\@firstoftwo
    \fi\fi}
```

`nolol` is just a key here. We'll use it below, of course.

```
\lst@Key{nolol}{false}[t]{\lstKV@SetIf{#1}\lst@ifnolol}
\def\lst@nololtrue{\let\lst@ifnolol\iftrue}
\let\lst@ifnolol\iffalse % init
```

18.1 Floats, boxes and captions

`captionpos`
`abovecaptionskip`
`belowcaptionskip`
`label`
`title`
`caption`

Some keys and ...

```
\lst@Key{captionpos}{t}{\def\lst@captionpos{#1}}
\lst@Key{abovecaptionskip}\smallskipamount{%
  \def\lst@abovecaption{#1}}
\lst@Key{belowcaptionskip}\smallskipamount{%
  \def\lst@belowcaption{#1}}
```

Rolf Niepraschk proposed `title`.

```
\lst@Key{label}\relax{\def\lst@label{#1}}
\lst@Key{title}\relax{\def\lst@title{#1}\let\lst@caption\relax}
\lst@Key{caption}\relax{\lstKV@OptArg[{#1}]{#1}%
    {\def\lst@caption{##2}\def\lst@@caption{##1}}%
     \let\lst@title\@empty}
\lst@AddToHookExe{TextStyle}
    {\let\lst@caption\@empty \let\lst@@caption\@empty
     \let\lst@title\@empty \let\lst@label\@empty}
```

`\thelstlisting`
`\lstlistingname`
`\lstlistingnamestyle`
`numberbychapter`

... and how the caption numbers look like. I switched to `\@ifundefined` (instead of `\ifx \@undefined`) after an error report from Denis Girou.

This is set `\AtBeginDocument` so that the user can specify whether or not the counter should be reset at each chapter before the counter is defined, using the `numberbychapter` key.

```
\AtBeginDocument{
  \@ifundefined{thechapter}{\let\lst@ifnumberbychapter\iffalse}{}
  \lst@ifnumberbychapter
      \newcounter{lstlisting}[chapter]
      \gdef\thelstlisting%
           {\ifnum \c@chapter>\z@ \thechapter.\fi \@arabic\c@lstlisting}
  \else
      \newcounter{lstlisting}
      \gdef\thelstlisting{\@arabic\c@lstlisting}
  \fi}
```

```
\lst@UserCommand\lstlistingname{Listing}
\lst@UserCommand\lstlistingnamestyle{}
```

```
\lst@Key{numberbychapter}{true}[t]{%
  \lstKV@SetIf{#1}\lst@ifnumberbychapter}
```

`\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`.

```
\@ifundefined{abovecaptionskip}{%
  \newskip\abovecaptionskip%
  \newskip\belowcaptionskip%
}{}
\@ifundefined{@makecaption}{%
  \long\def\@makecaption#1#2{%
    \vskip\abovecaptionskip%
    \sbox\@tempboxa{#1: #2}%
    \ifdim \wd\@tempboxa >\hsize%
        #1: #2\par%
    \else%
        \global \@minipagefalse%
        \hb@xt@\hsize{\hfil\box\@tempboxa\hfil}%
    \fi%
   \vskip\belowcaptionskip}%
}{}
```

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. Karl Berry asked for the ability of customizing the label. So `\lstlistingnamestyle` was introduced in front of `\lstlistingname`.

```
\def\fnum@lstlisting{%
  {\lstlistingnamestyle\lstlistingname
    \ifx\lst@@caption\@empty\else~\thelstlisting\fi}%
}
```

Captions are set only for display style listings – thanks to Peter Löffler for reporting the bug and to Axel Sommerfeldt for analyzing the bug. We `\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.

```
\def\lst@MakeCaption#1{%
  \lst@ifdisplaystyle
    \ifx #1t \allowbreak%
        \ifx\lst@@caption\@empty\expandafter\lst@HRefStepCounter \else
                                \expandafter\refstepcounter
        \fi {lstlisting}%
        \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.

```
        \let\lst@arg\lst@intname \lst@ReplaceIn\lst@arg\lst@filenamerpl
        \global\let\lst@name\lst@arg \global\let\lstname\lst@name
        \lst@ifnolol\else
            \ifx\lst@@caption\@empty
                \ifx\lst@caption\@empty
                    \ifx\lst@intname\@empty \else \def\lst@temp{ }%
                    \ifx\lst@intname\lst@temp \else
                        \addcontentsline{lol}{lstlisting}\lst@name
                    \fi\fi
                \fi
            \else
                \addcontentsline{lol}{lstlisting}%
                    {\protect\numberline{\thelstlisting}\lst@@caption}%
            \fi
        \fi
    \fi
```

We make a caption if and only if the caption is not empty and the user requested a caption at $\texttt{\#1} \in \{\texttt{t}, \texttt{b}\}$. To disallow pagebreaks between caption (or title) and a listing, we redefine the primitive `\vskip` locally to insert `\nobreak`s. Note that we allow pagebreaks in front of a 'top-caption' and after a 'bottom-caption'. Also, the `\ignorespaces` in the `\@makecaption` call is added to match what LaTeX does in `\@caption`; the AMSbook class (and perhaps others) assume this is present and attempt to strip it off when testing for an empty caption, causing a bug noted by Xiaobo Peng.

To do: This redefinition is a brute force method. Is there a better one?

```
    \ifx\lst@caption\@empty\else
        \lst@IfSubstring #1\lst@captionpos
            {\begingroup \let\@@vskip\vskip
             \def\vskip{\afterassignment\lst@vskip \@tempskipa}%
             \def\lst@vskip{\nobreak\@@vskip\@tempskipa\nobreak}%
             \par\@parboxrestore\normalsize\normalfont % \noindent (AS)
             %%\ifx #1t\allowbreak \fi
             \ifx\lst@title\@empty
                 \lst@makecaption\fnum@lstlisting{%
                   \ignorespaces \lst@caption}
             \else
                 \lst@maketitle\lst@title % (AS)
             \fi
             \ifx #1b\allowbreak \fi
             \endgroup}{}%
    \fi
  \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` `\lst@maketitle` Axel proposed the first definition. The other two are default definitions. They may be adjusted to make listings compatible with other packages and classes.

```
\def\lst@makecaption{\@makecaption}
\def\lst@maketitle{\@makecaption\lst@title@dropdelim}
\def\lst@title@dropdelim#1{\ignorespaces}
```

The following caption(2) support comes also from Axel.

```
\AtBeginDocument{%
\@ifundefined{captionlabelfalse}{}{%
  \def\lst@maketitle{\captionlabelfalse\@makecaption\@empty}}%
\@ifundefined{caption@startrue}{}{%
  \def\lst@maketitle{\caption@startrue\@makecaption\@empty}}%
}
```

`\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`.

```
\def\lst@HRefStepCounter#1{%
    \begingroup
    \c@lstlisting\lst@neglisting
    \advance\c@lstlisting\m@ne \xdef\lst@neglisting{\the\c@lstlisting}%
    \ifx\hyper@refstepcounter\@undefined\else
        \hyper@refstepcounter{#1}%
    \fi
    \endgroup}
\gdef\lst@neglisting{\z@}% init
```

`boxpos` `\lst@boxtrue` sets the vertical alignment of the (possibly) used box respectively indicates that a box is used.

```
\lst@Key{boxpos}{c}{\def\lst@boxpos{#1}}
```

```
\def\lst@boxtrue{\let\lst@ifbox\iftrue}
\let\lst@ifbox\iffalse
```

`float` `floatplacement` Matthias Zenger asked for double-column floats, so I've inserted some code. We first check for a star …

```
\lst@Key{float}\relax[\lst@floatplacement]{%
```

```
    \lstKV@SwitchCases{#1}%
    {true:\let\lst@floatdefault\lst@floatplacement
          \let\lst@float\lst@floatdefault\\%
     false:\let\lst@floatdefault\relax
           \let\lst@float\lst@floatdefault
    }{\def\lst@next{\@ifstar{\let\lst@beginfloat\@dblfloat
                             \let\lst@endfloat\end@dblfloat
                             \lst@KFloat}%
                            {\let\lst@beginfloat\@float
                             \let\lst@endfloat\end@float
                             \lst@KFloat}}
      \edef\lst@float{#1}%
      \expandafter\lst@next\lst@float\relax}}
```

... and define `\lst@float`.

```
\def\lst@KFloat#1\relax{%
    \ifx\@empty#1\@empty
        \let\lst@float\lst@floatplacement
    \else
        \def\lst@float{#1}%
    \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.

```
\lst@Key{floatplacement}{tbp}{\def\lst@floatplacement{#1}}
\lst@AddToHook{PreSet}{\let\lst@float\lst@floatdefault}
\lst@AddToHook{TextStyle}{\let\lst@float\relax}
\let\lst@floatdefault\relax % init
```

`\lst@doendpe` is set according to `\lst@float` – thanks to Andreas Schmidt and Heiko Oberdiek.

```
\lst@AddToHook{DeInit}{%
    \ifx\lst@float\relax
        \global\let\lst@doendpe\@doendpe
    \else
        \global\let\lst@doendpe\@empty
    \fi}
```

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.

```
\AtBeginDocument{%
\@ifundefined{c@float@type}%
    {\edef\ftype@lstlisting{\ifx\c@figure\@undefined 1\else 4\fi}}
    {\edef\ftype@lstlisting{\the\c@float@type}%
      \addtocounter{float@type}{\value{float@type}}}%
}
```

18.2 Init and EOL

`aboveskip` `belowskip` We define and initialize these keys and prevent extra spacing for ‘inline’ listings (in particular if fancyvrb interface is active, problem reported by Denis Girou).

```
\lst@Key{aboveskip}\medskipamount{\def\lst@aboveskip{#1}}
\lst@Key{belowskip}\medskipamount{\def\lst@belowskip{#1}}
\lst@AddToHook{TextStyle}
```

```
    {\let\lst@aboveskip\z@ \let\lst@belowskip\z@}
```

`everydisplay` `\lst@ifdisplaystyle` Some things depend on display-style listings.

```
\lst@Key{everydisplay}{}{\def\lst@EveryDisplay{#1}}
\lst@AddToHook{TextStyle}{\let\lst@ifdisplaystyle\iffalse}
\lst@AddToHook{DisplayStyle}{\let\lst@ifdisplaystyle\iftrue}
\let\lst@ifdisplaystyle\iffalse
```

`\lst@Init` Begin a float or multicolumn environment if requested.

```
\def\lst@Init#1{%
    \begingroup
    \ifx\lst@float\relax\else
        \edef\@tempa{\noexpand\lst@beginfloat{lstlisting}[\lst@float]}%
        \expandafter\@tempa
    \fi
    \ifx\lst@multicols\@empty\else
        \edef\lst@next{\noexpand\multicols{\lst@multicols}}
        \expandafter\lst@next
    \fi
```

In restricted horizontal TeX mode we switch to `\lst@boxtrue`. In that case we make appropriate box(es) around the listing.

```
    \ifhmode\ifinner \lst@boxtrue \fi\fi
    \lst@ifbox
        \lsthk@BoxUnsafe
        \hbox to\z@\bgroup
             $\if t\lst@boxpos \vtop
        \else \if b\lst@boxpos \vbox
        \else \vcenter \fi\fi
        \bgroup \par\noindent
    \else
        \lst@ifdisplaystyle
            \lst@EveryDisplay
            \par\penalty-50\relax
            \vspace\lst@aboveskip
        \fi
    \fi
```

Moved `\vspace` after `\par`—or we can get an empty line atop listings. Bug reported by Jim Hefferon.

Now make the top caption.

```
    \normalbaselines
    \abovecaptionskip\lst@abovecaption\relax
    \belowcaptionskip\lst@belowcaption\relax
    \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` $\neq 0$.

```
    \lsthk@PreInit \lsthk@Init
    \lst@ifdisplaystyle
        \global\let\lst@ltxlabel\@empty
        \if@inlabel
            \lst@ifresetmargins
```

```
                \leavevmode
            \else
                \xdef\lst@ltxlabel{\the\everypar}%
                \lst@AddTo\lst@ltxlabel{%
                    \global\let\lst@ltxlabel\@empty
                    \everypar{\lsthk@EveryLine\lsthk@EveryPar}}%
            \fi
        \fi
        \everypar\expandafter{\lst@ltxlabel
                              \lsthk@EveryLine\lsthk@EveryPar}%
    \else
        \everypar{}\let\lst@NewLine\@empty
    \fi
    \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 `\lstenv@backslash`.

```
    \lst@Let{13}\lst@MProcessListing
    \let\lst@Backslash#1%
    \lst@EnterMode{\lst@Pmode}{\lst@SelectCharTable}%
    \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`.

```
\let\lst@InitFinalize\@empty % init
\lst@AddToHook{PreInit}
    {\rightskip\z@ \leftskip\z@ \parfillskip=\z@ plus 1fil
     \let\par\@@par}
\lst@AddToHook{EveryLine}{}% init
\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.

```
\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 and `\normalcolor` after a report by Walter E. Brown.

```
\def\lst@DeInit{%
    \lst@XPrintToken \lst@EOLUpdate
    \global\advance\lst@newlines\m@ne
    \lst@ifshowlines
        \lst@DoNewLines
    \else
        \setbox\@tempboxa\vbox{\lst@DoNewLines}%
    \fi
    \lst@ifdisplaystyle \par\removelastskip \fi
    \lsthk@ExitVars\everypar{}\lsthk@DeInit\normalbaselines\normalcolor
```

Place the bottom caption.

```
    \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.

```
    \lst@ifbox
        \egroup $\hss \egroup
        \vrule\@width\lst@maxwidth\@height\z@\@depth\z@
    \else
        \lst@ifdisplaystyle
            \par\penalty-50\vspace\lst@belowskip
        \fi
    \fi
```

End the multicolumn environment and/or float if necessary.

```
    \ifx\lst@multicols\@empty\else
        \def\lst@next{\global\let\@checkend\@gobble
                      \endmulticols
                      \global\let\@checkend\lst@@checkend}
        \expandafter\lst@next
    \fi
    \ifx\lst@float\relax\else
        \expandafter\lst@endfloat
    \fi
    \endgroup}
\let\lst@@checkend\@checkend
```

`\lst@maxwidth` is to be allocated, initialized and updated.

```
\newdimen\lst@maxwidth % \global
\lst@AddToHook{InitVars}{\global\lst@maxwidth\z@}
\lst@AddToHook{InitVarsEOL}
    {\ifdim\lst@currlwidth>\lst@maxwidth
         \global\lst@maxwidth\lst@currlwidth
     \fi}
```

`\lst@EOLUpdate` What do you think this macro does?

```
\def\lst@EOLUpdate{\lsthk@EOL \lsthk@InitVarsEOL}
```

`\lst@MProcessListing` This is what we have to do at EOL while processing a listing. We output all remaining characters and update the variables. If we've reached the last line, we check whether there is a next line interval to input or not.

```
\def\lst@MProcessListing{%
    \lst@XPrintToken \lst@EOLUpdate \lsthk@InitVarsBOL
    \global\advance\lst@lineno\@ne
    \ifnum \lst@lineno>\lst@lastline
        \lst@ifdropinput \lst@LeaveMode \fi
        \ifx\lst@linerange\@empty
            \expandafter\expandafter\expandafter\lst@EndProcessListing
        \else
            \lst@interrange
            \lst@GetLineInterval
            \expandafter\expandafter\expandafter\lst@SkipToFirst
        \fi
    \else
```

```
        \expandafter\lst@BOLGobble
    \fi}
```

`\lst@EndProcessListing` Default definition is `\endinput`. This works for `\lstinputlisting`.

```
\let\lst@EndProcessListing\endinput
```

`gobble` The key sets the number of characters to gobble each line.

```
\lst@Key{gobble}{0}{\def\lst@gobble{#1}}
```

`\lst@BOLGobble` If the number is positive, we set a temporary counter and start a loop.

```
\def\lst@BOLGobble{%
    \ifnum\lst@gobble>\z@
        \@tempcnta\lst@gobble\relax
        \expandafter\lst@BOLGobble@
\fi}
```

A nonpositive number terminates the loop (by not continuing). Note: This is not the macro just used in `\lst@BOLGobble`.

```
\def\lst@BOLGobble@@{%
    \ifnum\@tempcnta>\z@
        \expandafter\lst@BOLGobble@
    \fi}
```

If we gobble a backslash, we have to look whether this backslash ends an environment. Whether the coming characters equal e.g. `end{lstlisting}`, we either end the environment or insert all just eaten characters after the ‘continue loop’ macro.

```
\def\lstenv@BOLGobble@@{%
    \lst@IfNextChars\lstenv@endstring{\lstenv@End}%
    {\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.

```
\def\lst@BOLGobble@#1{%
    \let\lst@next#1%
    \ifx \lst@next\relax\else
    \ifx \lst@next\lst@MProcessListing\else
    \ifx \lst@next\lst@processformfeed\else
```

Otherwise we use one of the two submacros.

```
    \ifx \lst@next\lstenv@backslash
        \let\lst@next\lstenv@BOLGobble@@
    \else
        \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), . . .

```
        \ifx #1\lst@processtabulator
            \advance\@tempcnta-\lst@tabsize\relax
            \ifnum\@tempcnta<\z@
                \lst@length-\@tempcnta \lst@PreGotoTabStop
            \fi
```

. . . whereas any other character decreases the counter by one.

```
        \else
```

```
      \advance\@tempcnta\m@ne
    \fi
  \fi \fi \fi \fi
  \lst@next}
```

```
\def\lst@processformfeed{\lst@ProcessFormFeed}
\def\lst@processtabulator{\lst@ProcessTabulator}
```

18.3 List of listings

`name` Each pretty-printing command values `\lst@intname` before setting any keys.

`\lstname`
`\lst@name`
`\lst@intname`

```
\lst@Key{name}\relax{\def\lst@intname{#1}}
\lst@AddToHookExe{PreSet}{\global\let\lst@intname\@empty}
\lst@AddToHook{PreInit}{%
  \let\lst@arg\lst@intname \lst@ReplaceIn\lst@arg\lst@filenamerpl
  \global\let\lst@name\lst@arg \global\let\lstname\lst@name}
```

Use of `\lst@ReplaceIn` removes a bug first reported by Magne Rudshaug. Here is the replacement list.

```
\def\lst@filenamerpl{_\textunderscore $\textdollar -\textendash}
```

`\l@lstlisting` prints one 'lol' line.

```
\def\l@lstlisting#1#2{\@dottedtocline{1}{1.5em}{2.3em}{#1}{#2}}
```

`\lstlistlistingname` contains simply the header name.

```
\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.

```
\lst@UserCommand\lstlistoflistings{\bgroup
  \let\contentsname\lstlistlistingname
  \let\lst@temp\@starttoc \def\@starttoc##1{\lst@temp{lol}}%
  \tableofcontents \egroup}
```

For KOMA-script classes, we define it a la KOMA thanks to a bug report by Tino Langer. Markus Kohm suggested a much-improved version of this, which also works with the float package. The following few comments are from Markus.

Make use of `\float@listhead` if defined (e.g. using float or KOMA-Script)

```
\@ifundefined{float@listhead}{}{%
 \renewcommand*{\lstlistoflistings}{%
  \begingroup
```

Switch to one-column mode if the switch for switching is available.

```
   \@ifundefined{@restonecoltrue}{}{%
    \if@twocolumn
     \@restonecoltrue\onecolumn
    \else
     \@restonecolfalse
    \fi
   }%
   \float@listhead{\lstlistlistingname}%
```

Set `\parskip` to 0pt (should be!), `\parindent` to 0pt (better but not always needed), `\parfillskip` to 0pt plus 1fil (should be!).

```
      \parskip\z@\parindent\z@\parfillskip \z@ \@plus 1fil%
      \@starttoc{lol}%
```

Switch back to twocolumn (see above).

```
      \@ifundefined{@restonecoltrue}{}{%
        \if@restonecol\twocolumn\fi
      }%
    \endgroup
  }%
}
```

`\float@addtolists` The float package defines a generic way for packages to add things (such as chapter names) to all of the lists of floats other than the standard figure and table lists. Each package that defines a list of floats adds a command to `\float@addtolists`, and then packages (such as the KOMA-script document classes) which wish to add things to all lists of floats can then use it, without needing to be aware of all of the possible lists that could exist. Thanks to Markus Kohm for the suggestion.

Unfortunately, float defines this with `\newcommand`; thus, to avoid conflict, we have to redefine it after float is loaded. `\AtBeginDocument` is the easiest way to do this. Again, thanks to Markus for the advice.

```
\AtBeginDocument{%
  \@ifundefined{float@addtolists}%
    {\gdef\float@addtolists#1{\addtocontents{lol}{#1}}}%
    {\let\orig@float@addtolists\float@addtolists
     \gdef\float@addtolists#1{%
       \addtocontents{lol}{#1}%
       \orig@float@addtolists{#1}}}%
}%
```

18.4 Inline listings

18.4.1 Processing inline listings

`\lstinline` In addition to `\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.

```
\newcommand\lstinline[1][]{%
    \leavevmode\bgroup % \hbox\bgroup --> \bgroup
      \def\lst@boxpos{b}%
      \lsthk@PreSet\lstset{flexiblecolumns,#1}%
      \lsthk@TextStyle
      \@ifnextchar\bgroup{%
```

Luc Van Eycken reported, that the experimental implementation of `\lstinline` with braces instead of characters surrounding the source code resulted in an error if used in a tabular environment. He found that this error comes from the master counter (cf. appendix D (Dirty Tricks), item 5. (Brace hacks), of the TeXbook (p. 385-386)). Adding the following line at this point

```
%          \ifnum‘{=0}\fi%
%
```

remedies the wrong behaviour. But Qing Lee pointed out, that this breaks code like the one showed in 7.1 on 58 and proposed another solution which in turn broke the code needed by Luc:

```
% % \renewcommand\lstinline[1][]{%
% %   \leavevmode\bgroup % \hbox\bgroup --> \bgroup
% %   \def\lst@boxpos{b}%
% %   \lsthk@PreSet\lstset{flexiblecolumns,#1}%
% %   \lsthk@TextStyle
% %   \ifnum\iffalse{\fi‘}=\z@\fi
% %   \@ifnextchar\bgroup{%
% %     \ifnum‘{=\z@}\fi%
% %     \afterassignment\lst@InlineG \let\@let@token}{%
% %     \ifnum‘{=\z@}\fi\lstinline@}}
%
```

So finally the old code comes back and the people, who need a `\lstinline` with braces, should use the workaround from section 7.1 on page 58.

```
        \afterassignment\lst@InlineG \let\@let@token}%
                        \lstinline@}
\def\lstinline@#1{%
    \lst@Init\relax
    \lst@IfNextCharActive{\lst@InlineM#1}{\lst@InlineJ#1}}
\lst@AddToHook{TextStyle}{}% init
```

```
\lst@AddToHook{SelectCharTable}{\lst@inlinechars}
\global\let\lst@inlinechars\@empty
```

`\lst@InlineM` `\lst@InlineJ` treat the cases of ‘normal’ inlines and inline listings inside an argument. In the first case the given character ends the inline listing and EOL within such a listing immediately ends it and produces an error message.

```
\def\lst@InlineM#1{\gdef\lst@inlinechars{%
    \lst@Def{‘#1}{\lst@DeInit\egroup\global\let\lst@inlinechars\@empty}%
    \lst@Def{13}{\lst@DeInit\egroup \global\let\lst@inlinechars\@empty
        \PackageError{Listings}{lstinline ended by EOL}\@ehc}}%
    \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.

```
\def\lst@InlineJ#1{%
    \def\lst@temp##1#1{%
        \let\lst@arg\@empty \lst@InsideConvert{##1}\lst@arg
        \lst@DeInit\egroup}%
    \lst@temp}
```

`\lst@InlineG` is experimental.

```
\def\lst@InlineG{%
    \lst@Init\relax
    \lst@IfNextCharActive{\lst@InlineM\}}%
                        {\let\lst@arg\@empty \lst@InlineGJ}}
```

```
\def\lst@InlineGJ{\futurelet\@let@token\lst@InlineGJTest}
\def\lst@InlineGJTest{%
    \ifx\@let@token\egroup
        \afterassignment\lst@InlineGJEnd
        \expandafter\let\expandafter\@let@token
    \else
        \ifx\@let@token\@sptoken
            \let\lst@next\lst@InlineGJReadSp
        \else
            \let\lst@next\lst@InlineGJRead
        \fi
        \expandafter\lst@next
    \fi}
\def\lst@InlineGJEnd{\lst@arg\lst@DeInit\egroup}
\def\lst@InlineGJRead#1{%
    \lccode`\~=`#1\lowercase{\lst@lAddTo\lst@arg~}%
    \lst@InlineGJ}
\def\lst@InlineGJReadSp#1{%
    \lccode`\~=`\ \lowercase{\lst@lAddTo\lst@arg~}%
    \lst@InlineGJ#1}
```

18.4.2 Short inline listing environments

The implementation in this section is based on the shortvrb package, which is part of `doc.dtx` from the Standard LaTeX documentation package, version 2006/02/02 v2.1d. Portions of it are thus copyright 1993–2006 by The LaTeX3 Project and copyright 1989–1999 by Frank Mittelbach. Denis Bitouzé used the Corona crisis to have look at the error messages and found some typos.

`\lstMakeShortInline`
`\lstMakeShortInline@`

First, we supply an optional argument if it's omitted.

```
\newcommand\lstMakeShortInline[1][]{%
  \def\lst@shortinlinedef{\lstinline[#1]}%
  \lstMakeShortInline@}%
\def\lstMakeShortInline@#1{%
  \expandafter\ifx\csname lst@ShortInlineOldCatcode\string#1\endcsname\relax
    \lst@shortlstinlineinfo{Made }{#1}%
    \lst@add@special{#1}%
```

The character's current catcode is stored in `\lst@ShortInlineOldCatcode\`$\langle c\rangle$.

```
    \expandafter
    \xdef\csname lst@ShortInlineOldCatcode\string#1\endcsname{%
      \the\catcode`#1}%
```

The character is spliced into the definition using the same trick as used in `\verb` (for instance), having activated `~` in a group.

```
    \begingroup
      \catcode`\~\active  \lccode`\~`#1%
      \lowercase{%
```

The character's old meaning is recorded in `\lst@ShortInlineOldMeaning\`$\langle c\rangle$ prior to assigning it a new one.

```
        \global\expandafter\let
          \csname lst@ShortInlineOldMeaning\string#1\endcsname~%
          \expandafter\gdef\expandafter~\expandafter{%
            \lst@shortinlinedef#1}}%
```

```
    \endgroup
```

Finally the character is made active.

```
    \global\catcode`#1\active
```

If we suspect that ⟨*c*⟩ is already a short reference, we tell the user. Now he or she is responsible if anything goes wrong… (Change in listings: We give a proper error here.)

```
  \else
    \PackageError{Listings}%
    {\string\lstMakeShorterInline\ definitions cannot be nested}%
    {Use \string\lstDeleteShortInline first.}%
    {}%
  \fi}
```

`\lstDeleteShortInline`

```
\def\lstDeleteShortInline#1{%
  \expandafter\ifx%
      \csname lst@ShortInlineOldCatcode\string#1\endcsname\relax%
    \PackageError{Listings}%
    {#1 is not a short reference for \string\lstinline}%
    {Use \string\lstMakeShortInline first.}%
    {}%
  \else
    \lst@shortlstinlineinfo{Deleted }{#1 as}%
    \lst@rem@special{#1}%
    \global\catcode`#1\csname lst@ShortInlineOldCatcode\string#1\endcsname
    \global \expandafter\let%
      \csname lst@ShortInlineOldCatcode\string#1\endcsname \relax
    \ifnum\catcode`#1=\active
      \begingroup
        \catcode`\~\active  \lccode`\~`#1%
        \lowercase{%
          \global\expandafter\let\expandafter~%
          \csname lst@ShortInlineOldMeaning\string#1\endcsname}%
      \endgroup
    \fi
  \fi}
```

`\lst@shortlstinlineinfo`

```
\def\lst@shortlstinlineinfo#1#2{%
     \PackageInfo{Listings}{%
       #1\string#2 a short reference for \string\lstinline}}
```

`\lst@add@special` This helper macro adds its argument to the `\dospecials` macro which is conventionally used by verbatim macros to alter the catcodes of the currently active characters. We need to add `\do\`⟨*c*⟩ to the expansion of `\dospecials` after removing the character if it was already there to avoid multiple copies building up should `\lstMakeShortInline` not be balanced by `\lstDeleteShortInline` (in case anything that uses `\dospecials` cares about repetitions).

```
\def\lst@add@special#1{%
  \lst@rem@special{#1}%
  \expandafter\gdef\expandafter\dospecials\expandafter
    {\dospecials \do #1}%
```

Similarly we have to add `\@makeother\`⟨*c*⟩ to `\@sanitize` (which is used in things like ” to re-catcode all special characters except braces).

```
  \expandafter\gdef\expandafter\@sanitize\expandafter
    {\@sanitize \@makeother #1}}
```

`\lst@rem@special` The inverse of `\lst@add@special` is slightly trickier. `\do` is re-defined to expand to nothing if its argument is the character of interest, otherwise to expand simply to the argument. We can then re-define `\dospecials` to be the expansion of itself. The space after `=‘##1` prevents an expansion to `\relax`!

```
\def\lst@rem@special#1{%
  \def\do##1{%
    \ifnum‘#1=‘##1 \else \noexpand\do\noexpand##1\fi}%
  \xdef\dospecials{\dospecials}%
```

Fixing `\@sanitize` is the same except that we need to re-define `\@makeother` which obviously needs to be done in a group.

```
  \begingroup
    \def\@makeother##1{%
      \ifnum‘#1=‘##1 \else \noexpand\@makeother\noexpand##1\fi}%
    \xdef\@sanitize{\@sanitize}%
  \endgroup}
```

18.5 The input command

`\lst@MakePath` The macro appends a slash to a path if necessary.

`inputpath`

```
\def\lst@MakePath#1{\ifx\@empty#1\@empty\else\lst@MakePath@#1/\@nil/\fi}
\def\lst@MakePath@#1/{#1/\lst@MakePath@@}
\def\lst@MakePath@@#1/{%
    \ifx\@nil#1\expandafter\@gobble
          \else \ifx\@empty#1\else #1/\fi \fi
    \lst@MakePath@@}
```

Now we can empty the path or use `\lst@MakePath`.

```
\lst@Key{inputpath}{}{\edef\lst@inputpath{\lst@MakePath{#1}}}
```

`\lstinputlisting` inputs the listing or asks the user for a new file name.

```
\def\lstinputlisting{%
    \begingroup \lst@setcatcodes \lst@inputlisting}
\newcommand\lst@inputlisting[2][]{%
  \endgroup
```

`\lst@set` takes the local options, especially `inputpath=<path>` for formatting the input file. So `\lstset` must be updated to have the right value of `\lst@inputpath`. The whole procedure must be grouped to make the change local.

```
  \bgroup\def\lst@set{#1}%
  \expandafter\lstset\expandafter{\lst@set}%
  \IfFileExists{\lst@inputpath#2}%
    {\lst@InputListing{\lst@inputpath#2}}%
    {\filename@parse{\lst@inputpath#2}%
      \edef\reserved@a{\noexpand\lst@MissingFileError
        {\filename@area\filename@base}%
        {\ifx\filename@ext\relax tex\else\filename@ext\fi}}%
      \reserved@a
```

We must provide a valid value for `\lst@doendpe` in the (error) case that there exists no file.

```
  \let\lst@doendpe\@empty}%
  \egroup
  \lst@doendpe \@newlistfalse \ignorespaces%
}
```

We use `\lst@doendpe` to remove indention at the beginning of the next line—except there is an empty line after `\lstinputlisting`. Bug was reported by David John Evans and David Carlisle pointed me to the solution.

`\lst@MissingFileError` is a derivation of LaTeX's `\@missingfileerror`. The parenthesis have been added after Heiko Oberdiek reported about a problem discussed on TEX-D-L.

```
\def\lst@MissingFileError#1#2{%
    \typeout{^^J! Package Listings Error: File ‘#1(.#2)’ not found.^^J%
        ^^JType X to quit or <RETURN> to proceed,^^J%
        or enter new name. (Default extension: #2)^^J}%
    \message{Enter file name: }%
    {\endlinechar\m@ne \global\read\m@ne to\@gtempa}%
```

Typing `x` or `X` exits.

```
    \ifx\@gtempa\@empty \else
        \def\reserved@a{x}\ifx\reserved@a\@gtempa\batchmode\@@end\fi
        \def\reserved@a{X}\ifx\reserved@a\@gtempa\batchmode\@@end\fi
```

In all other cases we try the new file name.

```
        \filename@parse\@gtempa
        \edef\filename@ext{%
            \ifx\filename@ext\relax#2\else\filename@ext\fi}%
        \edef\reserved@a{\noexpand\IfFileExists %
                {\filename@area\filename@base.\filename@ext}%
            {\noexpand\lst@InputListing %
                {\filename@area\filename@base.\filename@ext}}%
            {\noexpand\lst@MissingFileError
                {\filename@area\filename@base}{\filename@ext}}}%
        \expandafter\reserved@a %
    \fi}
```

`\lst@ifdraft` makes use of `\lst@ifprint`. Enrico Straube requested the final option.

```
\let\lst@ifdraft\iffalse
\DeclareOption{draft}{\let\lst@ifdraft\iftrue}
\DeclareOption{final}{\let\lst@ifdraft\iffalse}
\lst@AddToHook{PreSet}
    {\lst@ifdraft
         \let\lst@ifprint\iffalse
         \@gobbletwo\fi\fi
     \fi}
```

`\lst@InputListing` The one and only argument is the file name, but we have the ‘implicit’ argument `\lst@set`. Note that `\lst@Init` takes `\relax` as argument.

```
\def\lst@InputListing#1{%
    \begingroup
      \lsthk@PreSet \gdef\lst@intname{#1}%
      \expandafter\lstset\expandafter{\lst@set}%
      \lsthk@DisplayStyle
      \catcode\active=\active
      \lst@Init\relax \let\lst@gobble\z@
      \lst@SkipToFirst
      \lst@ifprint \def\lst@next{\input{#1}}%
             \else \let\lst@next\@empty \fi
      \lst@next
      \lst@DeInit
    \endgroup}
```

The line `\catcode\active=\active`, which makes the CR-character active, has been added after a bug report by Rene H. Larsen.

`\lst@SkipToFirst` The end of line character either processes the listing or is responsible for dropping lines up to first printing line.

```
\def\lst@SkipToFirst{%
    \ifnum \lst@lineno<\lst@firstline
```

We drop the input and redefine the end of line characters.

```
        \lst@BeginDropInput\lst@Pmode
        \lst@Let{13}\lst@MSkipToFirst
        \lst@Let{10}\lst@MSkipToFirst
    \else
        \expandafter\lst@BOLGobble
    \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).

```
\def\lst@MSkipToFirst{%
    \global\advance\lst@lineno\@ne
    \ifnum \lst@lineno=\lst@firstline
        \lst@LeaveMode \global\lst@newlines\z@
        \lsthk@InitVarsBOL
        \lst@DisplayConsecutiveNumbersOrNot
        \expandafter\lst@BOLGobble
    \fi}
```

18.6 The environment

18.6.1 Low-level processing

`\lstenv@DroppedWarning` gives a warning if characters have been dropped.

```
\def\lstenv@DroppedWarning{%
    \ifx\lst@dropped\@undefined\else
        \PackageWarning{Listings}{Text dropped after begin of listing}%
    \fi}
\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`.

```
\begingroup \lccode`\~=`\^^M\lowercase{%
\gdef\lstenv@Process#1{%
    \ifx~#1%
```

We make no extra `\lstenv@ProcessM` definition since there is nothing to do at all if we've found an active EOL.

```
        \lstenv@DroppedWarning \let\lst@next\lst@SkipToFirst
    \else\ifx^^J#1%
        \lstenv@DroppedWarning \let\lst@next\lstenv@ProcessJ
    \else
        \let\lst@dropped#1\let\lst@next\lstenv@Process
    \fi \fi
    \lst@next}
}\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'.

```
\def\lstenv@ProcessJ{%
    \let\lst@arg\@empty
    \ifx\@currenvir\lstenv@name
        \expandafter\lstenv@ProcessJEnv
    \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`.

```
        \expandafter\def\expandafter\lst@temp\expandafter##1%
            \csname end\lstenv@name\endcsname
                {\lst@InsideConvert{##1}\lstenv@ProcessJ@}%
        \expandafter\lst@temp
    \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).

```
\begingroup \lccode`\~=`\\\lowercase{%
\gdef\lstenv@ProcessJ@{%
    \lst@lExtend\lst@arg
        {\expandafter\ \expandafter~\lstenv@endstring}%
    \catcode10=\active \lst@Let{10}\lst@MProcessListing
```

We execute `\lst@arg` to typeset the listing.

```
    \lst@SkipToFirst \lst@arg}
}\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.

```
\def\lstenv@ProcessJEnv#1\end#2{\def\lst@temp{#2}%
    \ifx\lstenv@name\lst@temp
        \lst@InsideConvert{#1}%
        \expandafter\lstenv@ProcessJ@
    \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.

```
        \lst@InsideConvert{#1\\end\{#2\}}%
        \expandafter\lstenv@ProcessJEnv
    \fi}
```

`\lstenv@backslash` Coming to a backslash we either end the listing or process a backslash and insert the eaten characters again.

```
\def\lstenv@backslash{%
    \lst@IfNextChars\lstenv@endstring
        {\lstenv@End}%
        {\expandafter\lsts@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.

```
\def\lstenv@End{%
    \ifx\@currenvir\lstenv@name
        \edef\lst@next{\noexpand\end{\lstenv@name}}%
    \else
        \def\lst@next{\csname end\lstenv@name\endcsname}%
    \fi
    \lst@next}
```

18.6.2 Defining new environments

`\lstnewenvironment` Now comes the main command. We define undefined environments only. On the parameter text `#1#2#` (in particular the last sharp) see the paragraph following example 20.5 on page 204 of 'The TEXbook'.

```
\lst@UserCommand\lstnewenvironment#1#2#{%
    \@ifundefined{#1}%
        {\let\lst@arg\@empty
         \lst@XConvert{#1}\@nil
         \expandafter\lstnewenvironment@\lst@arg{#1}{#2}}%
        {\PackageError{Listings}{Environment `#1' already defined}\@eha
         \@gobbletwo}}
\def\@tempa#1#2#3{%
  \gdef\lstnewenvironment@##1##2##3##4##5{%
    \begingroup
```

A lonely 'end environment' produces an error.

```
    \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.

```
    \global\@namedef{##2}{\def\lstenv@name{##2}%
        \begingroup \lst@setcatcodes \catcode\active=\active
        \csname##2@\endcsname}%
```

The submacro is defined via `\new@command`. We misuse `\l@ngrel@x` to make the definition `\global` and refine LATEX's `\@xargdef`.

```
    \let\l@ngrel@x\global
    \let\@xargdef\lstenv@xargdef
    \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):

```
        {\lsthk@PreSet ##4%
         \ifx\@currenvir\lstenv@name
             \def\lstenv@endstring{#1#2##1#3}%
         \else
             \def\lstenv@endstring{#1##1}%
         \fi
```

We redefine (locally) 'end environment' since ending is legal now. Note that the redefinition also works inside a TEX comment line.

```
         \@namedef{end##2}{\lst@DeInit ##5\endgroup
                          \lst@doendpe \@ignoretrue}%
```

`\lst@doendpe` again removes the indention 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.

```
         \lsthk@DisplayStyle
         \let\lst@EndProcessListing\lstenv@SkipToEnd
         \lst@Init\lstenv@backslash
         \lst@ifprint
             \expandafter\expandafter\expandafter\lstenv@Process
         \else
             \expandafter\lstenv@SkipToEnd
         \fi
         \lst@insertargs}%
    \endgroup}%
}
\let\lst@arg\@empty \lst@XConvert{end}\{\}\@nil
\expandafter\@tempa\lst@arg
\let\lst@insertargs\@empty
```

`\lstenv@xargdef` This is a derivation of LATEX's `\@xargdef`. We expand the submacro's name, use `\gdef` instead of `\def`, and hard code a kind of `\@protected@testopt`.

```
\def\lstenv@xargdef#1{
    \expandafter\lstenv@xargdef@\csname\string#1\endcsname#1}
\def\lstenv@xargdef@#1#2[#3][#4]#5{%
  \@ifdefinable#2{%
       \gdef#2{%
          \ifx\protect\@typeset@protect
            \expandafter\lstenv@testopt
          \else
            \@x@protect#2%
          \fi
          #1%
          {#4}}%
       \@yargdef
          #1%
           \tw@
           {#3}%
           {#5}}}
```

`\lstenv@testopt` The difference between this macro and `\@testopt` is that we temporaryly reset the catcode of the EOL character `^^M` to read the optional argument.

```
\long\def\lstenv@testopt#1#2{%
  \@ifnextchar[{\catcode\active5\relax \lstenv@testopt@#1}%
               {#1[{#2}]}}
\def\lstenv@testopt@#1[#2]{%
    \catcode\active\active
    #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.

```
\begingroup \lccode`\~=`\\\lowercase{%
\gdef\lstenv@SkipToEnd{%
    \long\expandafter\def\expandafter\lst@temp\expandafter##\expandafter
        1\expandafter~\lstenv@endstring{\lstenv@End}%
    \lst@temp}
}\endgroup
```

`\lstenv@Error` is called by a lonely ‘end environment’.

```
\def\lstenv@Error#1{\PackageError{Listings}{Extra \string\end#1}%
    {I'm ignoring this, since I wasn't doing a \csname#1\endcsname.}}
```

`\lst@TestEOLChar` Here we test for the two possible EOL characters.

```
\begingroup \lccode`\~=`\^^M\lowercase{%
\gdef\lst@TestEOLChar#1{%
    \def\lst@insertargs{#1}%
    \ifx ~#1\@empty \else
    \ifx^^J#1\@empty \else
        \global\let\lst@intname\lst@insertargs
        \let\lst@insertargs\@empty
    \fi \fi}
}\endgroup
```

`lstlisting` (*env.*) 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.

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

There is a problem with vertical space below a listing as pointed out by Jean-Yves Baudais. A similar problem arises with a listing beginning at the top of a `\paragraph` or at the beginning of an `example` environment. Jean-Yves provided a solution—`\let\if@nobreak\iffalse`—as has been discussed on `fr.comp.text.tex`. The assumption, that the problem vanishes if there is a top rule at the beginning of the listing or if `\leavevmode` introduces the listing, was wrong as Karl Berry and Sven Schreiber reported independently, so the proposed code goes into the second part of the environment definition. Enrico Gregorio answered on `https://tex.stackexchange.com/questions/489121/is-it-a-listings-package-bug` that the previous solution

`\let\if@nobreak\iffalse` is wrong because it is a local assignment, but a globally setting is needed.

```
    \@nobreakfalse
    \csname\@lst @SaveFirstNumber\endcsname%
  }
⟨/kernel⟩
```

19 Documentation support

`\begin{lstsample}[`⟨*point list*⟩`]{`⟨*left*⟩`}{`⟨*right*⟩`}`

`\end{lstsample}`

> Roughly speaking all material in between this environment is executed 'on the left side' and typeset verbatim on the right. ⟨*left*⟩ is executed before the left side is typeset, and similarly ⟨*right*⟩ before the right-hand side.
>
> ⟨*point list*⟩ is used as argument to the `point` key. This is a special key used to highlight the keys in the examples.

`\begin{lstxsample}{`⟨*point list*⟩`}`

`\end{lstxsample}`

> The material in between is (a) added to the left side of the next `lstsample` environment and (b) typeset verbatim using the whole line width.

`\newdocenvironment{`⟨*name*⟩`}{`⟨*short name*⟩`}{`⟨*begin code*⟩`}{`⟨*end code*⟩`}`

> The ⟨*name*⟩ environment can be used in the same way as 'macro'. The provided(!) definitions `\Print`⟨*short name*⟩`Name` and `\SpecialMain`⟨*short name*⟩`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*⟩`(`⟨*dialect*⟩`),`.

`\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. Stephan Hennig noted a bug where `\ifalgorithmic` conflicts with an update to `algorithmic.sty`, so this has been changed to `\ifalgorithmicpkg`.

```
⟨*doc⟩
\let\lstdoc@currversion\fileversion
\RequirePackage[writefile]{listings}[2004/09/07]
\newif\iffancyvrb \IfFileExists{fancyvrb.sty}{\fancyvrbtrue}{}
\newif\ifcolor \IfFileExists{color.sty}{\colortrue}{}
\lst@false
\newif\ifhyper
\@ifundefined{pdfoutput}
    {}
    {\ifnum\pdfoutput>\z@ \lst@true \fi}
\@ifundefined{VTeXversion}
    {}
    {\ifnum\OpMode>\z@ \lst@true \fi}
\lst@if \IfFileExists{hyperref.sty}{\hypertrue}{}\fi
\newif\ifalgorithmicpkg \IfFileExists{algorithmic.sty}{\algorithmicpkgtrue}{}
\newif\iflgrind \IfFileExists{lgrind.sty}{\lgrindtrue}{}
\iffancyvrb \RequirePackage{fancyvrb}\fi
\ifhyper \RequirePackage[colorlinks]{hyperref}\else
    \def\href#1{\texttt}\fi
\ifcolor \RequirePackage{color}\fi
\ifalgorithmicpkg \RequirePackage{algorithmic}\fi
\iflgrind \RequirePackage{lgrind}\fi
\RequirePackage{nameref}
\RequirePackage{url}
\renewcommand\ref{\protect\T@ref}
\renewcommand\pageref{\protect\T@pageref}
```

19.2 Environments for notes

`\lst@BeginRemark`
`\lst@EndRemark`

We begin with two simple definitions …

```
\def\lst@BeginRemark#1{%
    \begin{quote}\topsep0pt\let\small\footnotesize\small#1:}
\def\lst@EndRemark{\end{quote}}
```

`TODO` (*env.*)
`ALTERNATIVE` (*env.*)
`REMOVED` (*env.*)
`OLDDEF` (*env.*)

… used to define some environments.

```
\newenvironment{TODO}
    {\lst@BeginRemark{To do}}{\lst@EndRemark}
\newenvironment{ALTERNATIVE}
    {\lst@BeginRemark{Alternative}}{\lst@EndRemark}
\newenvironment{REMOVED}
    {\lst@BeginRemark{Removed}}{\lst@EndRemark}
\newenvironment{OLDDEF}
    {\lst@BeginRemark{Old definition}}{\lst@EndRemark}
```

`advise` (*env.*)
`\advisespace`

The environment uses `\@listi`.

```
\def\advise{\par\list\labeladvise
    {\advance\linewidth\@totalleftmargin
     \@totalleftmargin\z@
     \@listi
     \let\small\footnotesize \small\sffamily
     \parsep \z@ \@plus\z@ \@minus\z@
     \topsep6\p@ \@plus1\p@\@minus2\p@
     \def\makelabel##1{\hss\llap{##1}}}}
\let\endadvise\endlist
```

```
\def\advisespace{\hbox{}\qquad}
\def\labeladvise{$\to$}
```

`syntax` (*env.*)
`\syntaxbreak`
`\syntaxnewline`
`\syntaxor`

This environment uses `\list` with a special `\makelabel`, …

```
\newenvironment{syntax}
   {\list{}{\itemindent-\leftmargin
    \def\makelabel##1{\hss\lst@syntaxlabel##1,,,,\relax}}}
   {\endlist}
```

… which is defined here. The comma separated items are placed as needed.

```
\def\lst@syntaxlabel#1,#2,#3,#4\relax{%
    \llap{\scriptsize\itshape#3}%
    \def\lst@temp{#2}%
    \expandafter\lst@syntaxlabel@\meaning\lst@temp\relax
    \rlap{\hskip-\itemindent\hskip\itemsep\hskip\linewidth
          \llap{\ttfamily\lst@temp}\hskip\labelwidth
          \def\lst@temp{#1}%
          \ifx\lst@temp\lstdoc@currversion#1\fi}}
\def\lst@syntaxlabel@#1>#2\relax
    {\edef\lst@temp{\zap@space#2 \@empty}}
```

```
\newcommand*\syntaxnewline{\newline\hbox{}\kern\labelwidth}
\newcommand*\syntaxor{\qquad or\qquad}
\newcommand*\syntaxbreak
    {\hfill\kern0pt\discretionary{}{\kern\labelwidth}{}}
\let\syntaxfill\hfill
```

`\alternative` iterates down the list and inserts vertical rule(s).

```
\def\alternative#1{\lst@true \alternative@#1,\relax,}
\def\alternative@#1,{%
    \ifx\relax#1\@empty
        \expandafter\@gobble
    \else
        \ifx\@empty#1\@empty\else
            \lst@if \lst@false \else $\vert$\fi
            \textup{\texttt{#1}}%
        \fi
    \fi
    \alternative@}
```

19.3 Extensions to doc

`\m@cro@` We need a slight modification of doc's internal macro. The former argument #2 has become #3. This change is not marked below. The second argument is now ⟨*short name*⟩.

```
\long\def\m@cro@#1#2#3{\endgroup \topsep\MacroTopsep \trivlist
  \edef\saved@macroname{\string#3}%
  \def\makelabel##1{\llap{##1}}%
```

```
\if@inlabel
  \let\@tempa\@empty \count@\macro@cnt
  \loop \ifnum\count@>\z@
    \edef\@tempa{\@tempa\hbox{\strut}}\advance\count@\m@ne \repeat
  \edef\makelabel##1{\llap{\vtop to\baselineskip
                             {\@tempa\hbox{##1}\vss}}}%
  \advance \macro@cnt \@ne
\else  \macro@cnt\@ne  \fi
\edef\@tempa{\noexpand\item[%
   #1%
     \noexpand\PrintMacroName
   \else
```

The next line has been modified.

```
     \expandafter\noexpand\csname Print#2Name\endcsname % MODIFIED
   \fi
   {\string#3}]}%
\@tempa
\global\advance\c@CodelineNo\@ne
 #1%
    \SpecialMainIndex{#3}\nobreak
    \DoNotIndex{#3}%
 \else
```

Ditto.

```
    \csname SpecialMain#2Index\endcsname{#3}\nobreak % MODIFIED
 \fi
\global\advance\c@CodelineNo\m@ne
\ignorespaces}
```

`\macro`
`\environment`

These two definitions need small adjustments due to the modified `\m@cro@`.

```
\def\macro{\begingroup
   \catcode`\\12
   \MakePrivateLetters \m@cro@ \iftrue {Macro}}% MODIFIED
\def\environment{\begingroup
   \catcode`\\12
   \MakePrivateLetters \m@cro@ \iffalse {Env}}% MODIFIED
```

`\newdocenvironment`

This command simply makes definitions similar to ‘environment’ and provides the printing and indexing commands.

```
\def\newdocenvironment#1#2#3#4{%
    \@namedef{#1}{#3\begingroup \catcode`\\12\relax
                  \MakePrivateLetters \m@cro@ \iffalse {#2}}%
    \@namedef{end#1}{#4\endmacro}%
    \@ifundefined{Print#2Name}{\expandafter
        \let\csname Print#2Name\endcsname\PrintMacroName}{}%
    \@ifundefined{SpecialMain#2Index}{\expandafter
        \let\csname SpecialMain#2Index\endcsname\SpecialMainIndex}{}}
```

`aspect` (*env.*)
`\PrintAspectName`
`\SpecialMainAspectIndex`

The environment and its ‘print’ and ‘index’ commands.

```
\newdocenvironment{aspect}{Aspect}{}{}
\def\PrintAspectName#1{}
\def\SpecialMainAspectIndex#1{%
    \@bsphack
    \index{aspects:\levelchar\protect\aspectname{#1}}%
    \@esphack}
```

`lstkey` (*env.*) `\PrintKeyName` `\SpecialMainKeyIndex` One more environment with its ‘print’ and ‘index’ commands.

```
\newdocenvironment{lstkey}{Key}{}{}
\def\PrintKeyName#1{\strut\keyname{#1}\ }
\def\SpecialMainKeyIndex#1{%
    \@bsphack
    \index{keys\levelchar\protect\keyname{#1}}%
    \@esphack}
```

`\labelargcount` `macroargs` (*env.*) We just allocate a counter and use LaTeX’s `\list` to implement this environment.

```
\newcounter{argcount}
\def\labelargcount{\texttt{\#\arabic{argcount}}\hskip\labelsep$=$}
```

```
\def\macroargs{\list\labelargcount
    {\usecounter{argcount}\leftmargin=2\leftmargin
     \parsep \z@ \@plus\z@ \@minus\z@
     \topsep4\p@ \@plus\p@ \@minus2\p@
     \itemsep\z@ \@plus\z@ \@minus\z@
     \def\makelabel##1{\hss\llap{##1}}}}
\def\endmacroargs{\endlist\@endparenv}
```

19.4 The `lstsample` environment

`lstsample` (*env.*) We store the verbatim part and write the source code also to file.

```
\lst@RequireAspects{writefile}
```

```
\newbox\lst@samplebox
\lstnewenvironment{lstsample}[3][]
    {\global\let\lst@intname\@empty
     \gdef\lst@sample{#2}%
     \setbox\lst@samplebox=\hbox\bgroup
         \setkeys{lst}{language={},style={},tabsize=4,gobble=5,%
             basicstyle=\small\ttfamily,basewidth=0.51em,point={#1}}
         #3%
         \lst@BeginAlsoWriteFile{\jobname.tmp}}
    {\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.

```
     \ifdim \wd\lst@samplebox>.5\linewidth
         \begin{center}%
             \hbox to\linewidth{\box\lst@samplebox\hss}%
         \end{center}%
         \lst@sampleInput
     \else
         \begin{center}%
         \begin{minipage}{0.45\linewidth}\lst@sampleInput\end{minipage}%
         \qquad
         \begin{minipage}{0.45\linewidth}%
             \hbox to\linewidth{\box\lst@samplebox\hss}%
         \end{minipage}%
         \end{center}%
     \fi}
```

The new keyword class `point`.

```
\lst@InstallKeywords{p}{point}{pointstyle}\relax{keywordstyle}{}ld
```

`lstxsample` (*env.*) Omitting `\lst@EndWriteFile` leaves the file open.

```
\lstnewenvironment{lstxsample}[1][]
    {\begingroup
         \setkeys{lst}{belowskip=-\medskipamount,language={},style={},%
             tabsize=4,gobble=5,basicstyle=\small\ttfamily,%
             basewidth=0.51em,point={#1}}
         \lst@BeginAlsoWriteFile{\jobname.tmp}}
    {\endgroup
     \endgroup}
```

`\lst@sampleInput` inputs the ‘left-hand’ side.

```
\def\lst@sampleInput{%
    \MakePercentComment\catcode`\^^M=10\relax
    \small\lst@sample
    {\setkeys{lst}{SelectCharTable=\lst@ReplaceInput{\^\^I}%
                                  {\lst@ProcessTabulator}}%
     \leavevmode \input{\jobname.tmp}}\MakePercentIgnore}
```

19.5 Miscellaneous

Sectioning and cross referencing We begin with a redefinition paragraph.

```
\renewcommand\paragraph{\@startsection{paragraph}{4}{\z@}%
                                      {1.25ex \@plus1ex \@minus.2ex}%
                                      {-1em}%
                                      {\normalfont\normalsize\bfseries}}
```

We introduce `\lstref` which prints section number together with its name.

```
\def\lstref#1{\emph{\ref{#1} \nameref{#1}}}
```

Moreover we adjust the table of contents. The `\phantomsection` before adding the contents line provides hyperref with an appropriate destination for the contents line link, thereby ensuring that the contents line is at the right level in the PDF bookmark tree.

```
\def\@part[#1]#2{\ifhyper\phantomsection\fi
    \addcontentsline{toc}{part}{#1}%
    {\parindent\z@ \raggedright \interlinepenalty\@M
     \normalfont \huge \bfseries #2\markboth{}{}\par}%
    \nobreak\vskip 3ex\@afterheading}
\renewcommand*\l@section[2]{%
    \addpenalty\@secpenalty
    \addvspace{.25em \@plus\p@}%
    \setlength\@tempdima{1.5em}%
    \begingroup
      \parindent \z@ \rightskip \@pnumwidth
      \parfillskip -\@pnumwidth
      \leavevmode
      \advance\leftskip\@tempdima
      \hskip -\leftskip
      #1\nobreak\hfil \nobreak\hb@xt@\@pnumwidth{\hss #2}\par
    \endgroup}
\renewcommand*\l@subsection{\@dottedtocline{2}{0pt}{2.3em}}
\renewcommand*\l@subsubsection{\@dottedtocline{3}{0pt}{3.2em}}
```

Indexing The 'user' commands. There are two different ways to mark up a key. `\ikeyname` is the commmand for keys which are used *i*nline, `\rkeyname` defines the *r*eference of a key—displayed green—, `\rstyle` is defined below.

```
\newcommand\ikeyname[1]{%
    \lstkeyindex{#1}{}%
    \lstaspectindex{#1}{}%
    \keyname{#1}}
\newcommand\rkeyname[1]{%
    \@bsphack
    \lstkeyindex{#1}{}%
    \lstaspectindex{#1}{}%
    \@esphack{\rstyle\keyname{#1}}}
```

The same rules apply to `\icmdname` and `\rcmdname`:

```
\newcommand\icmdname[1]{%
    \@bsphack
    \lstaspectindex{#1}{}%
    \@esphack\texttt{\string#1}}
\newcommand\rcmdname[1]{%
    \@bsphack
    \lstaspectindex{#1}{}%
    \@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.

```
\def\lstaspectindex#1#2{%
    \global\@namedef{lstkandc@\string#1}{}%
    \@ifundefined{lstisaspect@\string#1}
        {\index{unknown\levelchar
                \protect\texttt{\protect\string\string#1}#2}}%
        {\index{\@nameuse{lstisaspect@\string#1}\levelchar
                \protect\texttt{\protect\string\string#1}#2}}%
}
\def\lstkeyindex#1#2{%
%    \index{key\levelchar\protect\keyname{#1}#2}%
}
```

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.

```
\def\lstisaspect[#1]#2{%
    \global\@namedef{lstaspect@#1}{#2}%
    \lst@AddTo\lst@allkeysandcmds{,#2}%
    \@for\lst@temp:=#2\do
    {\ifx\@empty\lst@temp\else
         \global\@namedef{lstisaspect@\lst@temp}{#1}%
     \fi}}
\gdef\lst@allkeysandcmds{}
```

This relation is also good to print all keys and commands of a particular aspect . . .

```
\def\lstprintaspectkeysandcmds#1{%
    \lst@true
    \expandafter\@for\expandafter\lst@temp
    \expandafter:\expandafter=\csname lstaspect@#1\endcsname\do
    {\lst@if\lst@false\else, \fi \texttt{\lst@temp}}}
```

. . . or to check the reference. Note that we've defined `\lstkandc@`⟨*name*⟩ in `\lstaspectindex`.

```
\def\lstcheckreference{%
    \@for\lst@temp:=\lst@allkeysandcmds\do
   {\ifx\lst@temp\@empty\else
        \@ifundefined{lstkandc@\lst@temp}
        {\typeout{\lst@temp\space not in reference guide?}}{}%
    \fi}}
```

Unique styles

```
\newcommand*\lst{\texttt{lst}}
\newcommand*\Cpp{C\texttt{++}}
\let\keyname\texttt
\let\keyvalue\texttt
\let\hookname\texttt
\newcommand*\aspectname[1]{{\normalfont\sffamily#1}}
```

```
\DeclareRobustCommand\packagename[1]{%
    {\leavevmode\text@command{#1}%
     \switchfontfamily\sfdefault\rmdefault
     \check@icl #1\check@icr
     \expandafter}}%
\renewcommand\packagename[1]{{\normalfont\sffamily#1}}
\def\switchfontfamily#1#2{%
    \begingroup\xdef\@gtempa{#1}\endgroup
    \ifx\f@family\@gtempa\fontfamily#2%
                     \else\fontfamily#1\fi
    \selectfont}
```

The color mainly for keys and commands in the reference guide—*r* means reference.

```
\ifcolor
    \definecolor{darkgreen}{rgb}{0,0.5,0}
    \def\rstyle{\color{darkgreen}}
\else
    \let\rstyle\empty
\fi
```

Commands for credits and helpers There are two commands for credits and helpers:

1. `\lstthanks` is used to put a name of a contributor into the section "Closing and credit". It has two arguments: `#1` is the name, `#2` the email address—the email addres is not shown.

2. `\lsthelper` must be used in the text to show the name of the helper (argument `#1`), the date of the contribution (argument `#2`) and a short text about the contribution (argument `#3`). Only the first argument is printed.

```
\gdef\lst@emails{}
\newcommand*\lstthanks[2]
    {#1\lst@AddTo\lst@emails{,#1,<#2>}%
     \ifx\@empty#2\@empty\typeout{Missing email for #1}\fi}
\newcommand*\lsthelper[3]
```

```
    {{\let~\ #1}%
     \lst@IfOneOf#1\relax\lst@emails
     {}{\typeout{^^JWarning: Unknown helper #1.^^J}}}
```

Languages and styles

```
\lstdefinelanguage[doc]{Pascal}{%
  morekeywords={alfa,and,array,begin,boolean,byte,case,char,const,div,%
     do,downto,else,end,false,file,for,function,get,goto,if,in,%
     integer,label,maxint,mod,new,not,of,or,pack,packed,page,program,%
     procedure,put,read,readln,real,record,repeat,reset,rewrite,set,%
     text,then,to,true,type,unpack,until,var,while,with,write,writeln},%
  sensitive=false,%
  morecomment=[s]{(*}{*)},%
  morecomment=[s]{\{}{\}},%
  morestring=[d]{'}}
```

```
\lstdefinestyle{}
    {basicstyle={},%
     keywordstyle=\bfseries,identifierstyle={},%
     commentstyle=\itshape,stringstyle={},%
     numberstyle={},stepnumber=1,%
     pointstyle=\pointstyle}
\def\pointstyle{%
    {\let\lst@um\@empty \xdef\@gtempa{\the\lst@token}}%
    \expandafter\lstkeyindex\expandafter{\@gtempa}{}%
    \expandafter\lstaspectindex\expandafter{\@gtempa}{}%
    \rstyle}
\lstset{defaultdialect=[doc]Pascal,language=Pascal,style={}}
```

19.6 Scanning languages

`\lstscanlanguages` We modify some internal definitions and input the files.

```
\def\lstscanlanguages#1#2#3{%
    \begingroup
        \def\lst@DefDriver@##1##2##3##4[##5]##6{%
           \lst@false
           \lst@lAddTo\lst@scan{##6(##5),}%
           \begingroup
           \@ifnextchar[{\lst@XDefDriver{##1}##3}{\lst@DefDriver@@##3}}%
        \def\lst@XXDefDriver[##1]{}%
        \lst@InputCatcodes
        \def\lst@dontinput{#3}%
        \let\lst@scan\@empty
        \lst@for{#2}\do{%
            \lst@IfOneOf##1\relax\lst@dontinput
               {}%
               {\InputIfFileExists{##1}{}{}}}%
        \global\let\@gtempa\lst@scan
    \endgroup
    \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.

```
\def\lstprintlanguages#1{%
    \def\do##1{\setbox\@tempboxa\hbox{##1\space\space}%
        \ifdim\wd\@tempboxa<.5\linewidth \wd\@tempboxa.5\linewidth
                                   \else \wd\@tempboxa\linewidth \fi
        \box\@tempboxa\allowbreak}%
    \begin{quote}
      \par\noindent
      \hyphenpenalty=\@M \rightskip=\z@\@plus\linewidth\relax
      \lst@BubbleSort#1%
      \expandafter\lst@NextLanguage#1\relax(\relax),%
    \end{quote}}
```

We get and define the current language and …

```
\def\lst@NextLanguage#1(#2),{%
    \ifx\relax#1\else
        \def\lst@language{#1}\def\lst@dialects{(#2),}%
        \expandafter\lst@NextLanguage@
    \fi}
```

… gather all available dialect of this language (note that the list has been sorted)

```
\def\lst@NextLanguage@#1(#2),{%
    \def\lst@temp{#1}%
    \ifx\lst@temp\lst@language
        \lst@lAddTo\lst@dialects{(#2),}%
        \expandafter\lst@NextLanguage@
    \else
```

or begin to print this language with all its dialects. Therefor we sort the dialects

```
        \do{\lst@language
        \ifx\lst@dialects\lst@emptydialect\else
            \expandafter\lst@NormedDef\expandafter\lst@language
                \expandafter{\lst@language}%
            \space(%
            \lst@BubbleSort\lst@dialects
            \expandafter\lst@PrintDialects\lst@dialects(\relax),%
            )%
        \fi}%
        \def\lst@next{\lst@NextLanguage#1(#2),}%
        \expandafter\lst@next
    \fi}
\def\lst@emptydialect{(),}
```

and print the dialect with appropriate commas in between.

```
\def\lst@PrintDialects(#1),{%
    \ifx\@empty#1\@empty empty\else
        \lst@PrintDialect{#1}%
    \fi
    \lst@PrintDialects@}
\def\lst@PrintDialects@(#1),{%
    \ifx\relax#1\else
        , \lst@PrintDialect{#1}%
        \expandafter\lst@PrintDialects@
    \fi}
```

Here we take care of default dialects.

```
\def\lst@PrintDialect#1{%
```

```
    \lst@NormedDef\lst@temp{#1}%
    \expandafter\ifx\csname\@lst dd@\lst@language\endcsname\lst@temp
        \texttt{\underbar{#1}}%
    \else
        \texttt{#1}%
    \fi}
```

19.7 Bubble sort

`\lst@IfLE` ⟨*string 1*⟩`\relax\@empty`⟨*string 2*⟩`\relax\@empty{`⟨*then*⟩`}`⟨*else*⟩. If ⟨*string 1*⟩ ≤ ⟨*string 2*⟩, we execute ⟨*then*⟩ and ⟨*else*⟩ otherwise. Note that this comparision is case insensitive.

```
\def\lst@IfLE#1#2\@empty#3#4\@empty{%
    \ifx #1\relax
        \let\lst@next\@firstoftwo
    \else \ifx #3\relax
        \let\lst@next\@secondoftwo
    \else
        \lowercase{\ifx#1#3}%
            \def\lst@next{\lst@IfLE#2\@empty#4\@empty}%
        \else
            \lowercase{\ifnum‘#1<‘#3}\relax
                \let\lst@next\@firstoftwo
            \else
                \let\lst@next\@secondoftwo
            \fi
        \fi
    \fi \fi
    \lst@next}
```

`\lst@BubbleSort` is in fact a derivation of bubble sort.

```
\def\lst@BubbleSort#1{%
    \ifx\@empty#1\else
        \lst@false
```

We ‘bubble sort’ the first, second, … elements and …

```
        \expandafter\lst@BubbleSort@#1\relax,\relax,%
```

… then the second, third, … elements until no elemets have been swapped.

```
        \expandafter\lst@BubbleSort@\expandafter,\lst@sorted
                                      \relax,\relax,%
        \let#1\lst@sorted
        \lst@if
            \def\lst@next{\lst@BubbleSort#1}%
            \expandafter\expandafter\expandafter\lst@next
        \fi
    \fi}
\def\lst@BubbleSort@#1,#2,{%
    \ifx\@empty#1\@empty
        \def\lst@sorted{#2,}%
        \def\lst@next{\lst@BubbleSort@@}%
    \else
        \let\lst@sorted\@empty
        \def\lst@next{\lst@BubbleSort@@#1,#2,}%
    \fi
    \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).

```
\def\lst@BubbleSort@@#1,#2,{%
    \ifx\relax#1\else
        \ifx\relax#2%
            \lst@lAddTo\lst@sorted{#1,}%
            \expandafter\expandafter\expandafter\lst@BubbleSort@@@
        \else
            \lst@IfLE #1\relax\@empty #2\relax\@empty
                          {\lst@lAddTo\lst@sorted{#1,#2,}}%
                {\lst@true \lst@lAddTo\lst@sorted{#2,#1,}}%
            \expandafter\expandafter\expandafter\lst@BubbleSort@@
        \fi
    \fi}
\def\lst@BubbleSort@@@#1\relax,{}
```

⟨/doc⟩

20 Interfaces to other programs

20.1 0.21 compatibility

Some keys have just been renamed.

⟨*0.21⟩

```
\lst@BeginAspect{0.21}

\lst@Key{labelstyle}{}{\def\lst@numberstyle{#1}}
\lst@Key{labelsep}{10pt}{\def\lst@numbersep{#1}}
\lst@Key{labelstep}{0}{%
    \ifnum #1=\z@ \KV@lst@numbers{none}%
            \else \KV@lst@numbers{left}\fi
    \def\lst@stepnumber{#1\relax}}
\lst@Key{firstlabel}\relax{\def\lst@firstnumber{#1\relax}}
\lst@Key{advancelabel}\relax{\def\lst@advancenumber{#1\relax}}
\let\c@lstlabel\c@lstnumber
\lst@AddToHook{Init}{\def\thelstnumber{\thelstlabel}}
\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.

```
\lst@Key{first}\relax{\def\lst@firstline{#1\relax}}
\lst@Key{last}\relax{\def\lst@lastline{#1\relax}}

\lst@Key{framerulewidth}{.4pt}{\def\lst@framerulewidth{#1}}
\lst@Key{framerulesep}{2pt}{\def\lst@rulesep{#1}}
\lst@Key{frametextsep}{3pt}{\def\lst@frametextsep{#1}}
\lst@Key{framerulecolor}{}{\lstKV@OptArg[]{#1}%
    {\ifx\@empty##2\@empty
         \let\lst@rulecolor\@empty
     \else
        \ifx\@empty##1\@empty
```

```
             \def\lst@rulecolor{\color{##2}}%
         \else
             \def\lst@rulecolor{\color[##1]{##2}}%
         \fi
     \fi}}
\lst@Key{backgroundcolor}{}{\lstKV@OptArg[]{#1}%
    {\ifx\@empty##2\@empty
         \let\lst@bkgcolor\@empty
     \else
         \ifx\@empty##1\@empty
             \def\lst@bkgcolor{\color{##2}}%
         \else
             \def\lst@bkgcolor{\color[##1]{##2}}%
         \fi
     \fi}}
\lst@Key{framespread}{\z@}{\def\lst@framespread{#1}}
\lst@AddToHook{PreInit}
    {\@tempdima\lst@framespread\relax \divide\@tempdima\tw@
     \edef\lst@framextopmargin{\the\@tempdima}%
     \let\lst@framexrightmargin\lst@framextopmargin
     \let\lst@framexbottommargin\lst@framextopmargin
     \advance\@tempdima\lst@xleftmargin\relax
     \edef\lst@framexleftmargin{\the\@tempdima}}
```

Harald Harders had the idea of two spreads (inner and outer). We either divide the dimension by two or assign the two dimensions to inner- and outerspread.

```
\newdimen\lst@innerspread \newdimen\lst@outerspread
\lst@Key{spread}{\z@,\z@}{\lstKV@CSTwoArg{#1}%
    {\lst@innerspread##1\relax
     \ifx\@empty##2\@empty
         \divide\lst@innerspread\tw@\relax
         \lst@outerspread\lst@innerspread
     \else
         \lst@outerspread##2\relax
     \fi}}
\lst@AddToHook{BoxUnsafe}{\lst@outerspread\z@ \lst@innerspread\z@}
\lst@Key{wholeline}{false}[t]{\lstKV@SetIf{#1}\lst@ifresetmargins}
\lst@Key{indent}{\z@}{\def\lst@xleftmargin{#1}}
\lst@AddToHook{PreInit}
    {\lst@innerspread=-\lst@innerspread
     \lst@outerspread=-\lst@outerspread
     \ifodd\c@page \advance\lst@innerspread\lst@xleftmargin
             \else \advance\lst@outerspread\lst@xleftmargin \fi
     \ifodd\c@page
         \edef\lst@xleftmargin{\the\lst@innerspread}%
         \edef\lst@xrightmargin{\the\lst@outerspread}%
     \else
         \edef\lst@xleftmargin{\the\lst@outerspread}%
         \edef\lst@xrightmargin{\the\lst@innerspread}%
     \fi}
```

```
\lst@Key{defaultclass}\relax{\def\lst@classoffset{#1}}
\lst@Key{stringtest}\relax{}% dummy
\lst@Key{outputpos}\relax{\lst@outputpos#1\relax\relax}
```

```
\lst@Key{stringspaces}\relax[t]{\lstKV@SetIf{#1}\lst@ifshowstringspaces}
\lst@Key{visiblespaces}\relax[t]{\lstKV@SetIf{#1}\lst@ifshowspaces}
\lst@Key{visibletabs}\relax[t]{\lstKV@SetIf{#1}\lst@ifshowtabs}

\lst@EndAspect
⟨/0.21⟩
```

20.2 fancyvrb

Denis Girou asked whether fancyvrb and listings could work together.

`fancyvrb` We set the boolean and call a submacro.

```
⟨*kernel⟩
\lst@Key{fancyvrb}\relax[t]{%
    \lstKV@SetIf{#1}\lst@iffancyvrb
    \lstFV@fancyvrb}
\ifx\lstFV@fancyvrb\@undefined
    \gdef\lstFV@fancyvrb{\lst@RequireAspects{fancyvrb}\lstFV@fancyvrb}
\fi
⟨/kernel⟩
```

We end the job if fancyvrb is not present.

```
⟨*misc⟩
\lst@BeginAspect{fancyvrb}

\@ifundefined{FancyVerbFormatLine}
    {\typeout{^^J%
     ***^^J%
     *** `listings.sty' needs `fancyvrb.sty' right now.^^J%
     *** Please ensure its availability and try again.^^J%
     ***^^J}%
     \batchmode \@@end}{}
```

`\lstFV@fancyvrb` We assign the correct `\FancyVerbFormatLine` macro.

```
\gdef\lstFV@fancyvrb{%
    \lst@iffancyvrb
        \ifx\FancyVerbFormatLine\lstFV@FancyVerbFormatLine\else
            \let\lstFV@FVFL\FancyVerbFormatLine
            \let\FancyVerbFormatLine\lstFV@FancyVerbFormatLine
        \fi
    \else
        \ifx\lstFV@FVFL\@undefined\else
            \let\FancyVerbFormatLine\lstFV@FVFL
            \let\lstFV@FVFL\@undefined
        \fi
    \fi}
```

`\lstFV@VerbatimBegin` We initialize things if necessary.

```
\gdef\lstFV@VerbatimBegin{%
    \ifx\FancyVerbFormatLine\lstFV@FancyVerbFormatLine
        \lsthk@TextStyle \lsthk@BoxUnsafe
        \lsthk@PreSet
        \lst@activecharsfalse
        \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.

```
\xdef\lstFV@RestoreData{\noexpand\linewidth\the\linewidth\relax}%
        \lst@Init\relax
        \lst@ifresetmargins \advance\linewidth-\@totalleftmargin \fi
\lstFV@RestoreData
        \everypar{}\global\lst@newlines\z@
        \lst@mode\lst@nomode \let\lst@entermodes\@empty
        \lst@InterruptModes
```

Rolf Niepraschk reported a bug concerning ligatures to Denis Girou.

```
%% D.G. modification begin - Nov. 25, 1998
        \let\@noligs\relax
%% D.G. modification end
    \fi}
```

`\lstFV@VerbatimEnd` A box and macro must exist after `\lst@DeInit`. We store them globally.

```
\gdef\lstFV@VerbatimEnd{%
    \ifx\FancyVerbFormatLine\lstFV@FancyVerbFormatLine
        \global\setbox\lstFV@gtempboxa\box\@tempboxa
        \global\let\@gtempa\FV@ProcessLine
        \lst@mode\lst@Pmode
        \lst@DeInit
        \let\FV@ProcessLine\@gtempa
        \setbox\@tempboxa\box\lstFV@gtempboxa
        \par
    \fi}
```

The `\par` has been added after a bug report by Peter Bartke.

```
\newbox\lstFV@gtempboxa
```

We insert `\lstFV@VerbatimBegin` and `\lstFV@VerbatimEnd` where necessary.

```
\lst@AddTo\FV@VerbatimBegin\lstFV@VerbatimBegin
\lst@AddToAtTop\FV@VerbatimEnd\lstFV@VerbatimEnd
\lst@AddTo\FV@LVerbatimBegin\lstFV@VerbatimBegin
\lst@AddToAtTop\FV@LVerbatimEnd\lstFV@VerbatimEnd
\lst@AddTo\FV@BVerbatimBegin\lstFV@VerbatimBegin
\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.

```
\gdef\lstFV@FancyVerbFormatLine#1{%
    \let\lst@arg\@empty \lst@FVConvert#1\@nil
    \global\lst@newlines\z@
    \vtop{\noindent\lst@parshape
          \lst@ReenterModes
          \lst@arg \lst@PrintToken\lst@EOLUpdate\lsthk@InitVarsBOL
          \lst@InterruptModes}}
```

The `\lst@parshape` inside `\vtop` is due to a bug report from Peter Bartke. A `\leavevmode` became `\noindent`.

`fvcmdparams`
`morefvcmdparams`
These keys adjust `lst`@FVcmdparams, which will be used by the following conversion macro. The base set of commands and parameter numbers was provided by Denis Girou.

```
\lst@Key{fvcmdparams}%
    {\overlay\@ne}%
    {\def\lst@FVcmdparams{,#1}}
\lst@Key{morefvcmdparams}\relax{\lst@lAddTo\lst@FVcmdparams{,#1}}
```

`\lst@FVConvert` We do conversion or …

```
\gdef\lst@FVConvert{\@tempcnta\z@ \lst@FVConvertO@}%
\gdef\lst@FVConvertO@{%
    \ifcase\@tempcnta
        \expandafter\futurelet\expandafter\@let@token
        \expandafter\lst@FVConvert@@
    \else
```

… we append arguments without conversion, argument by argument, `\@tempcnta` times.

```
        \expandafter\lst@FVConvertO@a
    \fi}
\gdef\lst@FVConvertO@a#1{%
    \lst@lAddTo\lst@arg{{#1}}\advance\@tempcnta\m@ne
    \lst@FVConvertO@}%
```

Since `\@ifnextchar\bgroup` might fail, we have to use `\ifcat` here. Bug reported by Denis Girou. However we don't gobble space tokens as `\@ifnextchar` does.

```
\gdef\lst@FVConvert@@{%
    \ifcat\noexpand\@let@token\bgroup \expandafter\lst@FVConvertArg
                                \else \expandafter\lst@FVConvert@ \fi}
```

Coming to such a catcode = 1 character we convert the argument and add it together with group delimiters to `\lst@arg`. We also add `\lst@PrintToken`, which prints all collected characters before we forget them. Finally we continue the conversion.

```
\gdef\lst@FVConvertArg#1{%
    {\let\lst@arg\@empty
     \lst@FVConvert#1\@nil
     \global\let\@gtempa\lst@arg}%
     \lst@lExtend\lst@arg{\expandafter{\@gtempa\lst@PrintToken}}%
     \lst@FVConvert}
```

```
\gdef\lst@FVConvert@#1{%
    \ifx \@nil#1\else
       \if\relax\noexpand#1%
          \lst@lAddTo\lst@arg{\lst@OutputLostSpace\lst@PrintToken#1}%
       \else
          \lccode`\~=`#1\lowercase{\lst@lAddTo\lst@arg~}%
       \fi
       \expandafter\lst@FVConvert
    \fi}
```

Having no `\bgroup`, we look whether we've found the end of the input, and convert one token ((non)active character or control sequence).

```
\gdef\lst@FVConvert@#1{%
    \ifx \@nil#1\else
       \if\relax\noexpand#1%
          \lst@lAddTo\lst@arg{\lst@OutputLostSpace\lst@PrintToken#1}%
```

Here we check for registered commands with arguments and set the value of `\@tempcnta` as required.

```
          \def\lst@temp##1,#1##2,##3##4\relax{%
              \ifx##3\@empty \else \@tempcnta##2\relax \fi}%
          \expandafter\lst@temp\lst@FVcmdparams,#1\z@,\@empty\relax
       \else
          \lccode`\~=`#1\lowercase{\lst@lAddTo\lst@arg~}%
       \fi
       \expandafter\lst@FVConvert0@
    \fi}
```

```
\lst@EndAspect
⟨/misc⟩
```

20.3 Omega support

Ω support looks easy—I hope it works at least in some cases.

```
⟨*kernel⟩
```

```
\@ifundefined{ocp}{}
    {\lst@AddToHook{OutputBox}%
         {\let\lst@ProcessLetter\@firstofone
          \let\lst@ProcessDigit\@firstofone
          \let\lst@ProcessOther\@firstofone}}
```

```
⟨/kernel⟩
```

20.4 LGrind

`\lst@LGGetNames` is used to extract the language names from `\lst@arg` (the LGrind definition).

```
⟨*misc⟩
\lst@BeginAspect[keywords,comments,strings,language]{lgrind}
```

```
\gdef\lst@LGGetNames#1:#2\relax{%
    \lst@NormedDef\lstlang@{#1}\lst@ReplaceInArg\lstlang@{|,}%
    \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

```
\gdef\lst@LGGetValue#1{%
    \lst@false
    \def\lst@temp##1:#1##2##3\relax{%
        \ifx\@empty##2\else \lst@LGGetValue@{#1}\fi}
    \expandafter\lst@temp\lst@arg:#1\@empty\relax}
```

remove the pair if necessary.

```
\gdef\lst@LGGetValue@#1{%
    \lst@true
    \def\lst@temp##1:#1##2:##3\relax{%
        \@ifnextchar=\lst@LGGetValue@@{\lst@LGGetValue@@=}##2\relax
        \def\lst@arg{##1:##3}}%
    \expandafter\lst@temp\lst@arg\relax}
\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.

```
\gdef\lst@LGGetComment#1#2{%
    \let#2\@empty
    \lst@LGGetValue{#1b}%
    \lst@if
        \let#2\lst@LGvalue
        \lst@LGGetValue{#1e}%
        \ifx\lst@LGvalue\lst@LGEOL
            \edef\lstlang@{\lstlang@,commentline={#2}}%
            \let#2\@empty
        \else
            \edef#2{{#2}{\lst@LGvalue}}%
        \fi
    \fi}
```

`\lst@LGGetString` does the same for string delimiters, but it doesn't 'return' any value.

```
\gdef\lst@LGGetString#1#2{%
    \lst@LGGetValue{#1b}%
    \lst@if
        \let#2\lst@LGvalue
        \lst@LGGetValue{#1e}%
        \ifx\lst@LGvalue\lst@LGEOL
            \edef\lstlang@{\lstlang@,morestringizer=[l]{#2}}%
        \else
```

we must check for `\e`, i.e. whether we have to use `d`oubled or `b`ackslashed stringizer.

```
            \ifx #2\lst@LGvalue
                \edef\lstlang@{\lstlang@,morestringizer=[d]{#2}}%
            \else
                \edef\lst@temp{\lst@LGe#2}%
                \ifx \lst@temp\lst@LGvalue
                    \edef\lstlang@{\lstlang@,morestringizer=[b]{#2}}%
                \else
                    \PackageWarning{Listings}%
                    {String #2...\lst@LGvalue\space not supported}%
                \fi
            \fi
        \fi
    \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 `:`.

```
\gdef\lst@LGDefLang{%
```

```
    \lst@LGReplace
    \let\lstlang@\empty
```

Get the keywords and values of friends.

```
    \lst@LGGetValue{kw}%
    \lst@if
        \lst@ReplaceInArg\lst@LGvalue{{ },}%
        \edef\lstlang@{\lstlang@,keywords={\lst@LGvalue}}%
    \fi
    \lst@LGGetValue{oc}%
    \lst@if
        \edef\lstlang@{\lstlang@,sensitive=f}%
    \fi
    \lst@LGGetValue{id}%
    \lst@if
        \edef\lstlang@{\lstlang@,alsoletter=\lst@LGvalue}%
    \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.

```
    \lst@LGGetComment a\lst@LGa
    \lst@LGGetComment c\lst@LGc
    \ifx\lst@LGa\@empty
        \ifx\lst@LGc\@empty\else
            \edef\lstlang@{\lstlang@,singlecomment=\lst@LGc}%
        \fi
    \else
        \ifx\lst@LGc\@empty
            \edef\lstlang@{\lstlang@,singlecomment=\lst@LGa}%
        \else
            \edef\lstlang@{\lstlang@,doublecomment=\lst@LGc\lst@LGa}%
        \fi
    \fi
```

Now we parse the stringizers.

```
    \lst@LGGetString s\lst@LGa
    \lst@LGGetString l\lst@LGa
```

We test for the continuation capability and

```
    \lst@LGGetValue{tc}%
    \lst@if
        \edef\lstlang@{\lstlang@,lgrindef=\lst@LGvalue}%
    \fi
```

define the language.

```
    \expandafter\xdef\csname\@lst LGlang@\lst@language@\endcsname
        {\noexpand\lstset{\lstlang@}}%
```

Finally we inform the user of all ignored capabilities.

```
    \lst@ReplaceInArg\lst@arg{{: :}:}\let\lst@LGvalue\@empty
    \expandafter\lst@LGDroppedCaps\lst@arg\relax\relax
    \ifx\lst@LGvalue\@empty\else
        \PackageWarningNoLine{Listings}{Ignored capabilities for
            \space `\lst@language@' are\MessageBreak\lst@LGvalue}%
    \fi}
```

`\lst@LGDroppedCaps` just drops a previous value and appends the next capabilty name to `\lst@LGvalue`.

```
\gdef\lst@LGDroppedCaps#1:#2#3{%
    \ifx#2\relax
        \lst@RemoveCommas\lst@LGvalue
    \else
        \edef\lst@LGvalue{\lst@LGvalue,#2#3}%
        \expandafter\lst@LGDroppedCaps
    \fi}
```

`\lst@LGReplace` `\lst@LGe` We replace ‘escaped `:^$|`’ by catcode 11 versions, and other strings by some kind of short versions (which is necessary to get the above definitions work).

```
\begingroup
\catcode‘\/=0
\lccode‘\z=‘\:\lccode‘\y=‘\^\lccode‘\x=‘\$\lccode‘\v=‘\|
\catcode‘\\=12\relax
/lowercase{%
/gdef/lst@LGReplace{/lst@ReplaceInArg/lst@arg
    {{\:}{z }{\^}{y}{\$}{x}{\|}{v}{ \ }{ }{:\ :}{:}{\ }{ }{\(}({\)})}}
/gdef/lst@LGe{\e}
}
/endgroup
```

`\lst@LGRead` reads one language definition and defines the language if the correct one is found.

```
\gdef\lst@LGRead#1\par{%
    \lst@LGGetNames#1:\relax
    \def\lst@temp{endoflanguagedefinitions}%
    \ifx\lstlang@\lst@temp
        \let\lst@next\endinput
    \else
        \expandafter\lst@IfOneOf\lst@language@\relax\lstlang@
            {\lst@LGDefLang \let\lst@next\endinput}%
            {\let\lst@next\lst@LGRead}%
    \fi
    \lst@next}
```

`lgrindef` We only have to request the language and

```
\lst@Key{lgrindef}\relax{%
    \lst@NormedDef\lst@language@{#1}%
    \begingroup
    \@ifundefined{lstLGlang@\lst@language@}%
        {\everypar{\lst@LGRead}%
         \catcode‘\\=12\catcode‘\{=12\catcode‘\}=12\catcode‘\%=12%
         \catcode‘\#=14\catcode‘\$=12\catcode‘\^=12\catcode‘\_=12\relax
         \input{\lstlgrindeffile}%
        }{}%
    \endgroup
```

select it or issue an error message.

```
    \@ifundefined{lstLGlang@\lst@language@}%
        {\PackageError{Listings}%
         {LGrind language \lst@language@\space undefined}%
         {The language is not loadable. \@ehc}}%
        {\lsthk@SetLanguage
         \csname\@lst LGlang@\lst@language@\endcsname}}
```

`\lstlgrindeffile` contains just the file name.

```
\@ifundefined{lstlgrindeffile}
    {\lst@UserCommand\lstlgrindeffile{lgrindef.}}{}
```

```
\lst@EndAspect
⟨/misc⟩
```

20.5 hyperref

```
⟨*misc⟩
\lst@BeginAspect[keywords]{hyper}
```

`hyperanchor`
`hyperlink` determine the macro to set an anchor and a link, respectively.

```
\lst@Key{hyperanchor}\hyper@@anchor{\let\lst@hyperanchor#1}
\lst@Key{hyperlink}\hyperlink{\let\lst@hyperlink#1}
```

Again, the main thing is a special working procedure. First we extract the contents of `\lst@token` and get a free macro name for this current character string (using prefix `lstHR@` and a number as suffix). Then we make this free macro equivalent to `\@empty`, so it is not used the next time.

```
\lst@InstallKeywords{h}{hyperref}{}\relax{}
    {\begingroup
         \let\lst@UM\@empty \xdef\@gtempa{\the\lst@token}%
     \endgroup
     \lst@GetFreeMacro{lstHR@\@gtempa}%
     \global\expandafter\let\lst@freemacro\@empty
```

`\@tempcnta` is the suffix of the free macro. We use it here to refer to the last occurence of the same string. To do this, we redefine the output macro `\lst@alloverstyle` to set an anchor . . .

```
     \@tempcntb\@tempcnta \advance\@tempcntb\m@ne
     \edef\lst@alloverstyle##1{%
         \let\noexpand\lst@alloverstyle\noexpand\@empty
         \noexpand\smash{\raise\baselineskip\hbox
             {\noexpand\lst@hyperanchor{lst.\@gtempa\the\@tempcnta}%
                                       {\relax}}}%
```

. . . and a link to the last occurence (if there is any).

```
         \ifnum\@tempcnta=\z@ ##1\else
             \noexpand\lst@hyperlink{lst.\@gtempa\the\@tempcntb}{##1}%
         \fi}%
    }
    od
```

```
\lst@EndAspect
⟨/misc⟩
```

21 Epilogue

```
⟨*kernel⟩
```

Each option adds the aspect name to `\lst@loadaspects` or removes it from that data macro.

```
\DeclareOption*{\expandafter\lst@ProcessOption\CurrentOption\relax}
\def\lst@ProcessOption#1#2\relax{%
    \ifx #1!%
        \lst@DeleteKeysIn\lst@loadaspects{#2}%
    \else
        \lst@lAddTo\lst@loadaspects{,#1#2}%
    \fi}
```

The following aspects are loaded by default.

```
\@ifundefined{lst@loadaspects}
  {\def\lst@loadaspects{strings,comments,escape,style,language,%
      keywords,labels,lineshape,frames,emph,index}%
  }{}
```

We load the patch file, ...

```
\InputIfFileExists{lstpatch.sty}{}{}
```

... process the options, ...

```
\let\lst@ifsavemem\iffalse
\DeclareOption{savemem}{\let\lst@ifsavemem\iftrue}
\DeclareOption{noaspects}{\let\lst@loadaspects\@empty}
\ProcessOptions
```

... and load the aspects.

```
\lst@RequireAspects\lst@loadaspects
\let\lst@loadaspects\@empty
```

If present we select the empty style and language.

```
\lst@UseHook{SetStyle}\lst@UseHook{EmptyStyle}
\lst@UseHook{SetLanguage}\lst@UseHook{EmptyLanguage}
```

Finally we load the configuration files. Ulrike Fischer pointed out that this must happen with the correct catcode. At the moment the catcode of `^^M` is 9, which is wrong. So we reset the catcodes to the correct values before loading the files:

```
\lst@RestoreCatcodes%
\InputIfFileExists{listings.cfg}{}{}
\InputIfFileExists{lstlocal.cfg}{}{}
⟨info⟩\lst@ReportAllocs
⟨/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

1.0β 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

1.3 2004/09/07
  - another bugfix-release with LPPL-1.3-compliance

1.4 2007/02/26

- many bugfixes, and new maintainership
- several new and updated language definitions
- many small documentation improvements
- new keys, multicharacter string delimiters, short inline listings, and more.

1.5 2013/06/27

- new maintainership

1.6 2015/05/05

- add discussion about using `\lstinline[`⟨*key=value list*⟩`]{`⟨*source code*⟩`}`
- add section "Bugs and workarounds".

1.7 2018/09/02

- some new or updated language definitions
- several error corrections

1.8 from 2019/02/27 on

- corrected and activated the option `inputpath`
- some new or updated language definitions
- several error corrections
- introduced `\lstlistingnamestyle`

1.9 from 2023/02/27 on

- hopefully corrected the long outstanding wrong behaviour of displaying visible spaces
- generalized the use of `linerange`
- introduced key `consecutivenumbers`
- a pagebreak between a (top) caption and source code isn't allowed anymore
- the configuration files `listings.cfg` and `lstlocal.cfg` are read with the correct catcode
- some documentation and layout enhancements.

References

[DS13] Marco Daniel and Elke Schubert. The mdframed package, 2013.

[Fai11] Robin Fairbairns. The moreverb package, 2011.

[Mi04] Frank Mittelbach, Michel Goossens, Johannes Braams, David P. Carlisle, and Chris Rowley. The LaTeX-Companion, 2004.

[Som11] Axel Sommerfeldt. Customizing captions of floating environments, 2011.

Index