The LGrind package*

Various Artists

1997/01/30

Abstract

The LGrind package is a pretty printer for source code. It evolved from vgrind, supported TeX (tgrind) and LaTeX and now finally LaTeX 2_ε, in particular NFSS.

1 Introduction

1.1 What is it?

The LGrind package consists of three files:

- `lgrind` or `lgrind.exe` is the executable. It will convert an LaTeX-File with embedded listings or a source code file into a lot of macro-calls.
- `lgrind.sty` provides the environments and macros to make the produced mess readable.
- `lgrindef` contains the information needed to tell keywords from comments, comments from strings ...

1.2 Who is to blame?

LGrind is not the work of a single one. The program is based on vgrind by Dave Presotto & William Joy of UC Berkeley.

Van Jacobson wrote tgrind for TeX. Jerry Leichter of Yale University modified it for LaTeX. George V. Reilly of Brown University changed the name to lgrind and added the program-text-within-comments and @-within-LaTeX features, and finally Michael Piefel of Humboldt-University At Berlin converted it to work under LaTeX 2_ε, i. e. with NFSS, and improved the documentation. The latter is also the current maintainer of the package. If you encounter any problems or have ideas on how to improve the package, contact `piefel@cs.tu-berlin.de`.

However, there have been many contributors who supported the development; in particular the range of supported languages is mainly due to them. Unfortunately I do not know all of them, but my thanks go to everybody. A special Thank You to Torsten Schütze for his OS/2 support and many and various hints.

*This file has version number v3.4, last revised 1997/01/30.

2 LGrind – grind nice program listings

lgrind [-e] [-i] [-] [-n] [-c] [-t ⟨*width*⟩] [-h ⟨*header*⟩]
[-d[!] ⟨*description file*⟩] [-l ⟨*language*⟩] [-s] [⟨*name*⟩]

LGrind processes its input file(s) and writes the result to standard output. This output can be saved for later editing, inclusion in a larger document, etc.

The options are:

-e	process a LaTeX-file for embedded text.
-i	process for inclusion in a LaTeX-document.
-	take input from standard input.
-n	don't boldface keywords.
-c	don't treat @, etc. specially in comments.
-t	change tab width (default 8).
-h	specifies text to go on the left side of every output page (default is none).
-d	specifies the language definitions file.
-d!	same as -d, except the change is permanent (modifies executable)
-l	specifies the language to use.
-s	shows a list of currently known languages.

If neither -e nor -i are specified, a complete LaTeX-file is produced. When no language is given on the command line, LGrind tries to figure out the language via the extension of the file. A table of extensions and their languages is in the definition file. If the extension is unknown, C is chosen as a default.

When LGrind is started without any parameters, it will show a short help screen. The same happens when the appropriate option is given, but this is implementation dependent (usually what is common for the operation systemx, the default for DOS is `-?` and for UNIX `--help`).

The languages which are currently known are stored in the language definition file; their number increases more or less rapidly. At the time of writing the following languages are part of the distribution:

Ada	LaTeX[a]	RATFOR
Asm	LDL	RLaB[b]
Asm68	Linda	Russell
BASIC[a]	MASM[a]	SAS
Batch[b]	MATLAB[a]	sh
C	Emacs Mock Lisp	src
C++	model	SQL
csh	Modula2	Tcl/Tk[c]
FORTRAN	Pascal	VisualBasic[a]
Icon	PERL	VMSasm
ISP	PROLOG	yacc
Java	PostScript	

[a]John Leis, University of Southern Queensland, leis@usq.edu.au.
[b]Jim Green, University of Sheffield, j.j.green@shef.ac.uk
[c]Alexander Bednarz, Forsch.-zentr. Jülich, A.Bednarz@kfa-juelich.de

2.1 Usage

For example, to include a C-file named `foo.c` into your LaTeX-document, first give the command: `lgrind -i -lc foo.c > foo.tex`

This will generate `foo.tex`, which will have the pretty-printed version of `foo.c` with a lot of LaTeX-commands. Then include `lgrind.sty` as you include any other package, namely with `\usepackage{lgrind}` at the beginning of your LaTeX-document. Having done this, within the document you can include `foo.tex` using one of the commands described in the next section.

Use `lgrind -ly bary.y > bary.tex` to produce a stand-alone LaTeX-file from, say, a Yacc file. This uses Piet van Oostrum's fancyhdr.sty to make the headers and footers. BTW: You really should have this package. It's marvellous.

Finally, when you have a LaTeX-file with embedded program listings, you can preprocess it: `lgrind -e pretty-sources.lg > even-prettier-sources.tex` and get a new LaTeX-file which you then feed into LaTeX.

3 Using the LGrind.sty-file

The LGrind package is included via the `\usepackage` command. Currently the following options are supported:

procnames prints the names of starting and, if nested procedures are allowed, continued procedures in the margin. Don't make the margin too small, or don't make the names too long . . .

noindent cancels the indentation. Useful for long listings or listings within their own sections.

fussy lets LaTeX print all overfull hboxes. The default is to suppress this for about a tenth of an inch.

norules lets LGrind suppess the surrounding rules for included material (using `\lagrind` and `\lgrindfile`).

nolineno doesn't print line numbers.[1]

lineno5 prints line numbers every 5 lines. The default is 10.

leftnum print line numbers in the left margin. Default is the right.

3.1 Stand-alone and included listings

After processing a source code file with LGrind without the -e or -i options you get a LaTeX-file which can be directly compiled.

[1]To be exact, prints line numbers every 50,000 lines. But source code should never get so long in a single file – that's over 3 MByte! If you really want no numbers, set `\LGnuminterval` to zero; then you won't get procedure names, either.

When using -i LGrind will produce a file which can be included with the following macros:

`\lgrindfile` The first is `\lgrindfile{`⟨*file*⟩`}`, which will simply include the file ⟨*file*⟩ at that point of text, and will draw horizontal lines before and after the listing.

`\lagrind` The macro `\lagrind[`⟨*float*⟩`]{`⟨*file*⟩`}{`⟨*caption*⟩`}{`⟨*label*⟩`}` will put the listing also within a figure environment, using the ⟨*float*⟩ options (h, t, b or p), ⟨*caption*⟩ and ⟨*label*⟩ you gave. The starred form of `\lagrind` will also use the starred `figure*`.

3.2 Formatting your source code

Well, LGrind uses a different font for comments. This has as a consequence that functions you refer to are typeset differently in the program and in the comments, which is unsatisfactory. And, wouldn't it be great to use LaTeX commands to produce e. g. '©'?

The `lgrindef`-file defines environments for exactly these purposes. They are usually defined as follows, but of course it is possible to use other strings if the standard collides with the syntax of the language in question.

`%% %%` Text which is surrounded by `%%` is directly passed to LaTeX, a pair of curled braces around it. So the copyright symbol would be produced with

`%$ $%` `%%\copyright%%`. The `%$`⟨*text*⟩`$%` works much the same, except that ⟨*text*⟩ is set in math mode.

`%| |%` In `%|`⟨*text*⟩`|%` a kind of verbatim environment is provided. ⟨*Text*⟩ is typeset in typewriter.

`@ @` Program text within a comment is surrounded by `@`. The text is processed exactly as if it wasn't a comment. To produce an at-sign you have to use `@@`.

3.3 Embedded programs within a LaTeX-file

You don't have to process every single source file with LGrind, only to include it your document. Within the text of your LaTeX-file, you can mark groups of lines as program code, either text- or display-style to be specific. You can use several commands for controlling the inclusion of source code into your LaTeX-file.

Write your text, don't forget to include LGrind.sty. Use the following commands. You can 'debug' your text without including the lengthy listings. As a last step (but one), you process your file with LGrind and its option -e, which will provide you with your final LaTeX source file.

`%( )%` The commands are similar to the math environments. With `%(` and `%)` you
`@ @` obtain code in text style, i. e. in the same line. Surrounding the text with `@` is a short hand.

```
The expression
%(
a + 3
%)
produces 10.
```

produces the same as `The expression @a + 3@ produces 10.` The output will have 'a + 3' set as a program.

`%[ ]%` As with math, the square bracket equivalent produces display style listings, i. e. indented text on an own line.

`%*` As long listings tend not to fit on one page, there will be page breaks inserted. Since page breaks can considerably affect readability there will be none at all unless you insert lines consisting of just `%*`. Pages will end here and only here, but not necessarily here. (That is, you allow (or recommend) a page break. It will be taken if needed.)

`%=` You can insert your own code by using a line starting with `%=` in the program text. Whatever you enter after that is left in the output, exactly as you typed it. It will be executed in a strange environment, so doing anything fancy is very tricky.

`\Line` A macro, `\Line`, is provided to help you do simple things. For example,

```
%[
%=\Line{________\vdots}
        a = 1;
%]
```

produces:

$$\vdots$$
$$a = 1;$$

(Within the program text, `_` is active and expands to a fixed-width space. A whole bunch of macros are also defined. If you understand how lgrind sets lines up, you can replace the 8 `_`s with a call to `\Tab` (but I'll let you hang yourself on that one.))

`%<` The `%<`⟨*file*⟩ command includes ⟨*file*⟩ as a program listing in your document. Before inclusion it will be pretty printed. This is the almost the same as LGrinding the ⟨*file*⟩ separately and with -i and including it via `\lgrindfile`, only that it's
`%!` simpler for you. With `%!`⟨*command*⟩ the input is taken from a shell command.

While you can specify the language used on the command line, this does not
`%#` suffice for mixed-language programs (or projects). The command `%#`⟨*language*⟩ provides you a means to change the language on the fly wherever you want.

Important rules:

- `%` and the following character must be the first two characters on the line to be recognized.
- Put *nothing* on the line after the `%` and the key character. If you do that, lgrind will provide a default environment that will produce an `\hbox` for `%( )%`, and a `\vbox` for `%[ %]`. If you put stuff on the line, lgrind assumes you want to control the format completely. Doing this requires understanding *exactly* what the code lgrind produces is doing. (Sometimes I'm not sure I do!)
- `%)` and `%]` are, if I remember right, simply ignored outside of a code group, but any extra `%(` or `%[` produces a warning, so a missing `%)` or `%]` is usually caught.

- Remember that the code between `%(`/`%[` and `%)`/`%]` is put into a single box. Expect the usual problems with long boxes!

3.4 A final word

The output of lgrind always contains exactly one output line for each input line. Hence, you can look up line numbers in TEX error messages in your original file, rather than in the LGrinded (LGround?) file. (Of course, if the problem is in the lgrind output...)

The environment that lgrind's output builds uses a *lot* of stack space. I found I had to build a LATEX-with a larger stack, but that's not always necessary (it depends on exactly how you nest stuff.)

Many things are controllable by re-defining various macros. You can change what fonts lgrind will use for various kinds of things, how much it indents the output, whether it adds line numbers, and if so at what interval it prints them and whether it sticks them on the left or right, and so on. This stuff is all described below, though probably not very well. The default settings produce output that looks reasonable to me, though I can't say I'm ecstatic about it. Doing a *really* good job would require defining some special fonts.

4 The `lgrindef`-file

The `lgrindef` file is LGrind's language definition data base. It is here where LGrind learns what are keywords, what comments, where are functions, how to distinguish plain comments from LATEX-commands etc.

The first field is just the language name (and any variants of it). Thus the C language could be specified to LGrind as 'c' or 'C'.

4.1 Capabilities

Capabilities are of two types: Boolean capabilities which indicate that the language has some particular feature and string capabilities which give a regular expression or keyword list.

Entries may continue onto multiple lines by giving a \ as the last character of a line. Lines starting with `#` are comments.

The following table names and describes each capability.

ab Regular expression for the start of an alternate form comment

ae Regular expression for the end of an alternate form comment

bb Regular expression for the start of a block

be Regular expression for the end of a lexical block

cb Regular expression for the start of a comment

ce Regular expression for the end of a comment

cf (Boolean) Use specialized routine for detecting C functions

id String giving characters other than letters and digits that may legally occur in identifiers (default '_')

kw A list of keywords separated by spaces

lb Regular expression for the start of a character constant

le Regular expression for the end of a character constant

mb Regular expression for the start of TeX math within a comment

me Regular expression for the end of TeX math within a comment

np Regular expression for a line that does *not* contain the start of a procedure (e. g. prototypes)

oc (Boolean) Present means upper and lower case are equivalent

pb Regular expression for start of a procedure

pl (Boolean) Procedure definitions are constrained to the lexical level matched by the 'px' capability

px A match for this regular expression indicates that procedure definitions may occur at the next lexical level. Useful for lisp-like languages in which procedure definitions occur as subexpressions of defuns.

rb Regular expression for the start of block outside the actual code.[2]

sb Regular expression for the start of a string

se Regular expression for the end of a string

tb Regular expression for the start of TeX text within a comment

tc (String) Use the named language entry as a continuation of the current one

te Regular expression for the end of TeX text within a comment

tl (Boolean) Present means procedures are only defined at the top lexical level

vb Regular expression for the start of typewriter text within a comment

ve Regular expression for the end of typewriter text within a comment

zb Regular expression for the start of program text within a comment

ze Regular expression for the end of program text within a comment

[2] I included this especially for the `objects` and `records` in Pascal and Modula-2. They *end* (with the `<be>` expression), but shouldn't have influence on the surrounding procedure. When defining `record` as normal block start, its `end` ends the procedure. Workaround: Make `record` itself a procedure start. But that prints a continuation mark when `procnames` is on.

4.2 Regular Expressions

`lgrindef` uses regular expressions similar to those of ex and lex. The characters '^', '$', '|', ':', and '\' are reserved characters and must be 'quoted' with a preceding \ if they are to be included as normal characters.

The meta-symbols and their meanings are:

$ The end of a line

^ The beginning of a line

\d A delimiter (space, tab, newline, start of line)

\a Matches any string of symbols (like '.*' in lex)

\p Matches any identifier. In a procedure definition (the 'pb' capability) the string that matches this symbol is used as the procedure name.

( ) Grouping

| Alternation

? Last item is optional

\e Preceding any string means that the string will not match an input string if the input string is preceded by an escape character (\). This is typically used for languages (like C) that can include the string delimiter in a string by escaping it.

Unlike other regular expressions in the system, these match words and not characters. Hence something like '(tramp|steamer)flies?' would match 'tramp', 'steamer', 'trampflies', or 'steamerflies'. Contrary to some forms of regular expressions, `lgrindef` alternation binds very tightly. Grouping parentheses are likely to be necessary in expressions involving alternation.

4.3 Keyword List

The keyword list is just a list of keywords in the language separated by spaces. If the 'oc' boolean is specified, indicating that upper and lower case are equivalent, then all the keywords should be specified in lower case.

4.4 Configuration options

In addition to the language definitions the `lgrindef`-file contains various configuration data. When the entries do not exist, default values are used:

firstpreamble is the (LaTeX-)text that comes at the beginning of an stand alone file created by LGrind from source code.

filepreamble is inserted before every processed source file in a stand alone LaTeX-file. In these two preambles you can use `\f`, which will be substituted by the current input file (e. g. to put it into the header).

configuration follows the opening of the `lgrind`-environment. This is used for redefining the macros used within it, e. g. the fonts or the width of a space (the `\@ts` unit).

chartab is a list of characters that will be substituted by a LaTeX-string. This is useful when you do (or can) not use any of the fancy methods to persuade LaTeX into using your extended ASCII-characters. The format is a two digit hex number (the ASCII- (or whatever) value of the character), an equal sign, and the according LaTeX-string, ended with a colon. You have to escape certain characters (like the backslash). So if you, e. g., have IBM ASCII code page 437 input and use the german-package, you can have your ä using `84="a`. Note that the substituting string must contain more than one character; otherwise it will be ignored. To print a 'b' instead of an 'a' you can use `61={b}`.

5 The Implementation of LGrind.sty

```
⟨*package⟩
```

`\LGnuminterval` `\lc@unt` `\ln@xt`

The counter `\LGnuminterval` represents the line numbering interval. Its default is 10, it is set by two options and can be changed everywhere you want to. `\lc@unt` counts the current line, `\ln@xt` contains the next line to get numbered.

```
\newcount\lc@unt
\newcount\ln@xt
\newcount\LGnuminterval
\LGnuminterval=10
\DeclareOption{nolineno}{\LGnuminterval=50000}
\DeclareOption{lineno5}{\LGnuminterval=5}
```

`\LGleftnum`

Line numbers are usually on the right. By setting `LGleftnum` to true or false this behaviour can be altered. When putting the numbers on the left, procedure names will occur on the line *after* the start of the procedure; this is a bug. (Does anyone need it? Then tell me so.)

```
\newif\ifLGleftnum
\DeclareOption{leftnum}{\LGleftnumtrue}
```

`\LGindent`

`\LGindent` is the indentation for all display style listing lines.

```
\newskip\LGindent
\LGindent=1.6667\parindent
\DeclareOption{noindent}{\LGindent=0pt}
```

`\LGnorules`

Normally LGrind puts rules around everything that is included (via `\lagrind` and `\lgrindfile`), this can be changed with an option.

```
\newif\ifLGnorules
\DeclareOption{norules}{\LGnorulestrue}
```

`\LGsloppy`

`\LGsloppy` is the amount that a horizontal box may be overfull without getting a warning from LaTeX. This is useful since there are often many boxes which are overfull by only a few points, and this does not really show since listings are very ragged.

```
\newlength{\LGsloppy}
\setlength{\LGsloppy}{7.2pt}
\DeclareOption{fussy}{\LGsloppy=0pt}
```

`\Proc` `\ProcCont`

There's a `\Proc{`⟨*ProcName*⟩`}` at the start of each procedure. If the language allows nested procedures (e. g. Pascal), there will be a `\ProcCont{`⟨*ProcName*⟩`}` at the end of each inner procedure. (In this case, ⟨*ProcName*⟩ is the name of the outer procedure. I. e., `\ProcCont` marks the continuation of ⟨*ProcName*⟩).

`\DefaultProc` `\DefaultProcCont`

Default is not to do anything with the name. Optionally the names are printed in the same margin as the line numbers. The name is put into a box which will be output whenever it is not empty.

```
\newcommand{\DefaultProc}{\@gobble}
\newcommand{\DefaultProcCont}{\@gobble}
\DeclareOption{procnames}{
\renewcommand{\DefaultProc}[1]{\renewcommand{\Procname}{#1}%
\global\setbox\procbox=\hbox{\PNsize #1}}
\renewcommand{\DefaultProcCont}[1]{\renewcommand\Procname{#1}
\global\setbox\procbox=\hbox{\PNsize\dots #1}}}
\newbox\procbox
\newcommand{\Procname}{}
```

End of initialization, execute any options.

```
\ProcessOptions
```

`\BGfont` `\CMfont` `\NOfont` `\KWfont` `\STfont` `\VRfont` `\PNsize` `\LGsize` `\LGfsize`

These are the fonts and sizes for background (everything that doesn't fit elsewhere), comments, numbers, keywords, strings, variables, the procedure names in the margins, displayed code (`%[ ]%`), and included code (`\lgrindfile` and `\lagrind`), respectively. Note that the suffixes 'font' and 'size' have been chosen solely for the author's intention; you can do anything you want, e. g. `\tiny` comments. You have to use, however, font changes which don't require an argument.

```
\def\BGfont{\sffamily}
\def\CMfont{\rmfamily\itshape}
\def\NOfont{\sffamily}
\def\KWfont{\rmfamily\bfseries}
\def\STfont{\ttfamily}
\def\VRfont{\rmfamily}
\def\PNsize{\BGfont\small}
\def\LGsize{\small}
\def\LGfsize{\footnotesize}
```

`\ifLGinline` `\ifLGd@fault` `\LGbegin` `\LGend`

The flag `LGinline` is true for in-line code. `\LGbegin` and `\LGend` are default commands to open and close a code example and use it to perform certain ops depending whether we're in-line or display style.`\LGend` is a no-op unless `\LGbegin` (where `LGd@fault` is set true) was executed, so you can provide explicit open code on the `%[` or `%(` line without providing any special code on the matching `%]` or `%)` line.

```
\newif\ifLGinline
\newif\ifLGd@fault
\def\LGbegin{\ifLGinline$\hbox\else$$\vbox\fi\bgroup\LGd@faulttrue}
\def\LGend{\ifLGd@fault\egroup\ifLGinline$\else$$\fi\LGd@faultfalse\fi}
```

`\ifc@omment` `\ifstr@ng` These two conditions indicate if we are setting a comment or maybe a string constant, respectively.

```
\newif\ifc@mment
\newif\ifstr@ng
```

`\ifright@` To get decent quotes (opening and closing) within comments, we remember whether the next one is going to be '“' or, if true, '”'.

```
\newif\ifright@
```

`\ls@far` `\tb@x` `\TBw@d` These three are all for the sake of tabbing. `\ls@far` stores the “line so far”. The tabwidth goes in `\TBw@d`, whilst `\tb@x` is merely a temporary variable for `\Tab` and setting `\@ts`.

```
\newbox\ls@far
\newbox\tb@x
\newdimen\TBw@d
```

The underscore marks a point where the pre-processor wants a fixed-width space (of width `\@ts`).

```
\newdimen\@ts
{\catcode`\_=\active \gdef\@setunder{\let_=\sp@ce}}
```

`\lgrindhead` `\lgrindfilename` `\lgrindfilesize` `\lgrindmodyear` `\lgrindmodmonth` `\lgrindmodday` `\lgrindmodtime` We pollute the global namspace once more with these macros, for when they are used in the headers or footers, their values must still be known. Therefore they cannot be local to the `lgrind` environment.

```
\newcommand{\lgrindheader}{}
\newcommand{\lgrindfilename}{}\newcommand{\lgrindfilesize}{}
\newcommand{\lgrindmodyear}{}\newcommand{\lgrindmodmonth}{}
\newcommand{\lgrindmodday}{}\newcommand{\lgrindmodtime}{}
```

`lgrind` This is the environment that eventually defines all necessary macros for formatting. All LGrinded text goes into such an environment, no matter if directly so or from within another one. It takes one optional argument, the line number.

```
\newenvironment{lgrind}[1][1]{%
```

`\Line` The `\Line` macro is provided for use with `%=` in embedded listings. It's just there to hide the actual structure of this, for nobody *really* wants to know anyway.

```
\def\Line##1{\L{\LB{##1}}}%
```

\Head \File

The next are primarily meant for stand-alone listings. `\Head` and `\File` are inserted by LGrind, they define macros that contain a user-specified string (the header option -h), the name, size and modification time of the processed file. These can then be used e. g. in the headers and footers.

```
\newcommand{\Head}[1]{\gdef\lgrindhead{##1}}%
\newcommand{\File}[6]{\gdef\lgrindfilename{##1}\message{(LGround: ##1)}%
    \gdef\lgrindmodyear{##2}\gdef\lgrindmodmonth{##3}%
    \gdef\lgrindmodday{##4}\gdef\lgrindmodtime{##5}%
    \gdef\lgrindfilesize{##6}}%
```

The `\Proc`s now get what was specified for them in the options section.

```
\let\Proc=\DefaultProc%
\let\ProcCont=\DefaultProcCont%
```

We set a `\hfuzz` to prevent some of the lesser overfull hbox warnings.

```
\hfuzz=\LGsloppy
```

\NewPage

Each formfeed in the input is replaced by a `\NewPage` macro. If you really want a page break here, define this as `\vfill\eject`.

```
\def\NewPage{\filbreak\bigskip}%
```

\L

Each line of displayed program text is enclosed by a `\L{...}`. We turn each line into an hbox. Firstly we look whether we are in-line. Every `\LGnuminterval` lines we output a small line number in past the margin.

```
\ifLGinline
 \def\L##1{\setbox\ls@far\null{\CF\strut##1}\ignorespaces}%
```

\r@ghtlno \l@ftlno

Things get more difficult for display style listings. Here we set `\r@ghtlno` and `\l@ftlno` to no-ops, only to redefine them shortly after.

```
\else
 \let\r@ghtlno\relax\let\l@ftlno\relax
 \ifnum\LGnuminterval>\z@
  \ifLGleftnum
```

If there was a procedure name somewhere, `\procbox` is not empty and thus ready to be printed. Otherwise we test `\lc@unt` against `\ln@ext` to determine whether or not to print a line number.

```
   \def\l@ftlno{\ifvoid\procbox\ifnum\lc@unt>\ln@xt
     \global\advance\ln@xt by\LGnuminterval
     \llap{{\normalfont\scriptsize\the\lc@unt\quad}}\fi
     \else\llap{\box\procbox\quad}\fi}%
```

And once again when the line number is meant to be on the right.

```
  \else
   \def\r@ghtlno{\ifvoid\procbox\ifnum\lc@unt>\ln@xt
     \global\advance\ln@xt by\LGnuminterval
     \rlap{{\normalfont\scriptsize\enspace\the\lc@unt}}\fi
     \else\rlap{\enspace\box\procbox}\fi}%
  \fi
 \fi
```

\lc@unt is incremented and everything is squeezed into a \hbox.

```
\def\L##1{\@@par\setbox\ls@far=\null\strut
 \global\advance\lc@unt by1%
 \hbox to \hsize{\hskip\LGindent\l@ftlno ##1\egroup%
 \hfil\r@ghtlno}%
 \ignorespaces}%
\fi
```

The initialization of \lc@unt and \ln@xt. Every lgrind-environment starts over unless given a line number as argument.

```
\lc@unt=#1\advance\lc@unt by-1%
\ln@xt=\LGnuminterval\advance\ln@xt by-1%
\loop\ifnum\lc@unt>\ln@xt\advance\ln@xt by\LGnuminterval\repeat%
```

\LB \Tab

The following weirdness is to deal with tabs. "Pieces" of a line between tabs are output as \LB{...}. E. g., a line with a tab at column 16 would be output as \LB{xxx}\Tab{16}\LB{yyy}. (Actually, to reduce the number of characters in the .tex file the \Tab macro supplies the 2nd & subsequent \LBs.) We accumulate the \LB stuff in an \hbox. When we see a \Tab, we grab this hbox (using \lastbox) and turn it into a box that extends to the tab position. We stash this box in \ls@far & stick it on in front of the next piece of the line. (There must be a better way of doing tabs but I'm not enough of a TEXwizard to come up with it. Suggestions would be appreciated. Oh, well, this comment's been in here for a decade. I don't believe in Santa Claus.)

```
\def\LB{\hbox\bgroup\bgroup\box\ls@far\CF\let\next=}%
\def\Tab##1{\egroup\setbox\tb@x=\lastbox\TBw@d=\wd\tb@x%
 \advance\TBw@d by 1\@ts\ifdim\TBw@d>##1\@ts
  \setbox\ls@far=\hbox{\box\ls@far \box\tb@x \sp@ce}\else
  \setbox\ls@far=\hbox to ##1\@ts{\box\ls@far \box\tb@x \hfil}\fi\LB}%
```

A normal space is too thin for code listings. We make spaces & tabs be in \@ts units, which for displays are 80 % the width of a "0" in the typewriter font. For inline stuff, on the other hand, we prefer a somewhat smaller space – actually, the same size as normal inter-word spaces – to help make the included stuff look like a unit.

```
\ifLGinline\def\sp@ce{\hskip .3333em}%
\else \setbox\tb@x=\hbox{\texttt{0}}%
      \@ts=0.8\wd\tb@x \def\sp@ce{\hskip 1\@ts}\fi
\catcode`\_=\active \@setunder
```

\CF \N \K \V \ic@r \C \CE \S \SE

Font changing. Since we are usually changing the font inside of a \LB macro, we remember the current font in \CF & stick a \CF at the start of each new box. Also, the characters """ and "'" behave differently in comments than in code, and others behave differently in strings than in code.
\N is for numbers, \K marks keywords, \V variables, \C and \CE surround comments, \S and \SE strings. \ic@r inserts an optional \/.

```
\def\CF{\ifc@mment\CMfont\else\ifstr@ng\STfont\fi\fi}
```

```
\def\N##1{{\NOfont ##1}\global\futurelet\next\ic@r}%
\def\K##1{{\KWfont ##1}\global\futurelet\next\ic@r}%
\def\V##1{{\VRfont ##1}\global\futurelet\next\ic@r}%
\def\ic@r{\let\@tempa\/\ifx.\next\let\@tempa\relax%
 \else\ifx,\next\let\@tempa\relax\fi\fi\@tempa}%
\def\C{\egroup\bgroup\CMfont \global\c@mmenttrue \global\right@false}%
\def\CE{\egroup\bgroup \global\c@mmentfalse}%
\def\S{\egroup\bgroup\STfont \global\str@ngtrue}%
\def\SE{\egroup\bgroup \global\str@ngfalse}%
```

\, \! We need positive and negative thinspaces in both text and math modes, so we re-define \, and \! here. The definition for \, isn't really needed for LaTeX, but we try to be more complete. Note that in LaTeX terms, the new definition isn't robust, like the old – but nothing we produce here is likely to be robust – or *needs* to be! – anyway!

```
\def\,{\relax \ifmmode\mskip\thinmuskip \else\thinspace \fi}%
\def\!{\relax \ifmmode\mskip-\thinmuskip \else\negthinspace \fi}%
```

Special characters. \CH chooses its first option alone in math mode; its second option in a string; and its third option, enclosed in $s, otherwise. (At the moment, nothing is ever set in math mode, but you never know . . .)

```
\def\CH##1##2##3{\relax\ifmmode ##1\relax
\else\ifstr@ng ##2\relax\else$##3$\fi\fi }%
\def\{{\CH\lbrace {\char'173}\lbrace }%
\def\}{\CH\rbrace {\char'175}\rbrace }%
\def\1{\CH///}%                              % /
\def\2{\CH\backslash {\char'134}\backslash }%  % \
\def\|{\CH|{\char'174}|}%
\def\<{\CH<<<}%
\def\>{\CH>>>}%
\def\*{\CH***}\relax %\relax for DOCSTY
\def\-{\CH---}%
\def\_{\ifstr@ng {\char'137}\else
  \leavevmode \kern.06em \vbox{\hrule width.35em}%
  \ifdim\fontdimen\@ne\font=\z@ \kern.06em \fi\fi }%
\def\&{\textsf{\char'046}}%
\def\#{{\STfont\char'043}}%
\def\%{{\char'045}}%
\def\~{{\char'176}}%
\def\3{\ifc@mment\ifright@ ''\global\right@false%
                     \else``\global\right@true \fi
   \else{\texttt{\char'042}}\fi}%
\def\4{\ifc@mment'\else {\texttt{\char'015}}\fi}%
\def\5{{\texttt{\char'136}}}%
\def\${{\ifmmode\slshape\else\ifdim\fontdimen\@ne\font>\z@\slshape\fi\fi
  \char'044}}% %No $ in \it, use \sl
```

Finally we don't want any indentation other than our own. We allow LaTeX to stretch our listings a bit. Then we open a group, select the background font and (fanfare!) are ready to begin.

```
\parindent\z@\parskip\z@ plus 1pt\hsize\linewidth%
\bgroup\BGfont
}
```

This is the end of the `lgrind` environment. Rather short (in comparison!)

```
{\egroup\@@par}            % end of environment lgrind
```

The following are generated as part of opening and closing included code sequences.

```
\def\lgrinde{\ifLGinline\else\LGsize\fi\begin{lgrind}}
\def\endlgrinde{\end{lgrind}}
```

`\lagrind` The `lagrind` environment is one of two for including files. It puts its argument inside a `figure` environment. It can be used without or with a star (first line), and with or without the usual floating arguments (second and third).

```
\def\lagrind{\@ifstar{\@slagrind}{\@lagrind}}

\def\@lagrind{\@ifnextchar[{\@@lagrind}{\@@lagrind[t]}}
\def\@slagrind{\@ifnextchar[{\@@slagrind}{\@@slagrind[t]}}
```

`\@@lagrind` The unstarred version. Everything is pretty obvious, we open a `figure`, put in a `minipage`, input the file in question, make caption and label and that's it.

```
\def\@@lagrind[#1]#2#3#4{%
    \begin{figure}[#1]
\ifLGnorules\else\hrule\fi
\vskip .5\baselineskip
\begin{minipage}\columnwidth\LGsize\LGindent\z@
    \begin{lgrind}
\input #2\relax
    \end{lgrind}
\end{minipage}
\vskip .5\baselineskip plus .5\baselineskip
\ifLGnorules\else\hrule\fi\vskip .5\baselineskip
\begingroup
    \setbox\z@=\hbox{#4}%
    \ifdim\wd\z@>\z@
\caption{#3}%
\label{#4}%
    \else
\captcont{#3}%
    \fi
\endgroup
\vskip 2pt
    \end{figure}
}
```

`\@@slagrind` Déjà vu? The starred version got an asterisk attached to `figure`.

```
\def\@@slagrind[#1]#2#3#4{%
    \begin{figure*}[#1]
\ifLGnorules\else\hrule\fi
```

```
\vskip .5\baselineskip
\begin{minipage}\linewidth\LGsize\LGindent\z@
    \begin{lgrind}
\input #2\relax
    \end{lgrind}
\end{minipage}
\vskip .5\baselineskip plus .5\baselineskip
\ifLGnorules\else\hrule\fi\vskip .5\baselineskip
\begingroup
    \setbox\z@=\hbox{#4}%
    \ifdim\wd\z@>\z@
\caption{#3}%
\label{#4}%
    \else
\captcont{#3}%
    \fi
\endgroup
\vskip 2pt
    \end{figure*}
}
```

`\lgrindfile` This is similar. We draw lines above and below, no `figure`. But it can get longer than one page.

```
\def\lgrindfile#1{%
    \par\addvspace{0.1in}
    \ifLGnorules\else\hrule\fi
    \vskip .5\baselineskip
    \begingroup\LGfsize\LGindent\z@
\begin{lgrind}
    \input #1\relax
\end{lgrind}
    \endgroup
    \vskip .5\baselineskip
    \ifLGnorules\else\hrule\fi
    \addvspace{0.1in}
}
```

And now …

⟨/package⟩

That's it. Thank you for reading up to here.

Michael Piefel
piefel@cs.tu-berlin.de

Change History

Index

Numbers written in italic refer to the page where the corresponding entry is described, the ones underlined to the code line of the definition, the rest to the code lines where the entry is used.