The bashful Package*

Yossi Gil†
Department of Computer Science
The Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel

2011/03/12‡

Abstract

*It is sometimes useful to "*escape-to-shell*" from within LaTeX. The most obvious application is when the document explains something about the working of a computer program. Your text would be more robust to changes, and easier to write, if all the examples it gives, are run directly from within LaTeX.*

To facilitate this and other applications, package bashful *provides a convenient interface to TeX's primitive* `\write18`*—the execution of shell commands from within your input files, also known as* shell escape*. Text between* `\bash` *and* `\END` *is executed by* `bash`*, a popular Unix command line interpreter. Various flags control whether the executed commands and their output show up in the printed document, and whether they are saved to files.*

Although provisions are made for using shells other than `bash`*, this package may* not *operate without modifications on Microsoft's operating systems.*

Contents

†mailto:yogi@cs.Technion.ac.IL

‡This document describes bashful V 0.92.

1 Introduction

At the time I run this document through LaTeX, the temperature in Jerusalem, Israel, was *15° C*, while the weather condition was *clear*.

You may not care so much about these bits of truly ephemeral information, but you may be surprised that they were produced by the very process of LaTeXing the input.

How did I do that? Well, the first step was to write a series of shell commands that retrieve the current temperature, and another such series to obtain the current weather conditions. This task required connection to Google's weather service and minimal dexterity with Unix pipes and filters to process the output.

My command series to obtain the current temperature was:

```
% location=Jerusalem,Israel
server="http://www.Google.com/ig/api"
request="$server?weather=$location"
wget -q -O - $request |\
tr "<>" "\012\012" |\
grep temp_c |\
sed 's/[^0-9]//g'
```

while the weather condition was obtained by

```
% location=Jerusalem,Israel
server="http://www.Google.com/ig/api"
request="$server?weather=$location"
wget -q -O - $request |\
tr "<>" "\012\012" |\
grep "condition data" |\
head -n 1 |\
sed -e 's/^.*="//' -e 's/"\/*//' |\
tr 'A-Z' 'a-z'
```

The second step was coercing LaTeX to run these commands while processing my

document. To do that, I used package bashful,

```
\usepackage{bashful}
```

And, then, I wrapped each of these two series within a `\bash`...`\END` pair.

The `\bash` command, offered by this package, takes all subsequent lines, stopping at the closing `\END`, places these in a file, and then lets the `bash` shell interpreter execute this file.

Allowing LaTeX to run arbitrary shell commands can be dangerous—you never know whether that nice looking `.tex` file you received by email was prepared by a friend or a foe. This is the reason that you have to tell LaTeX explicitly that shell escapes are allowed. The `-shell-esc` flag does that. To process my document, I typed, at the command line,

```
% latex -shell-escape bashful.tex
```

What I actually wrote in the input to produce the temperature in Jerusalem, Israel was:

```
\bash[verbose,scriptFile=temperature.sh,stdoutFile=temperature.tex]
% location=Jerusalem,Israel
server="http://www.Google.com/ig/api"
request="$server?weather=$location"
wget -q -O - $request |\
tr "<>" "\012\012" |\
grep temp_c |\
sed 's/[^0-9]//g'
\END
```

The flags passed to the `bash` control sequence above instructed it:

1. to be verbose, typing out a detailed log of everything it did;
2. to save the shell commands in a script file named `temperature.sh`; and,
3. to store the standard output of the script in a file named `temperature.tex`.

To obtain the current weather condition in the capital I wrote:

```
\bash[verbose,scriptFile=condition.sh,stdoutFile=condition.tex]
% location=Jerusalem,Israel
server="http://www.Google.com/ig/api"
request="$server?weather=$location"
wget -q -O - $request |\
tr "<>" "\012\012" |\
grep "condition data" |\
head -n 1 |\
sed -e 's/^.*="//' -e 's/"\/*//' |\
tr 'A-Z' 'a-z'
\END
```

I wrote these two just after my `\begin{document}`. When LaTeX encountered these, it executed the bash commands and created two files `temperature.tex` and `condition.tex`.

Subsequently, I could use the content of these files by writing:

```
At the time I run this document through \LaTeX{},
  the temperature in Jerusalem, Israel,
  was~\emph{\input{temperature}\unskip\celsius},
  while the weather condition was
  \emph{\input{condition}}\unskip.

You may not care so much about these bits of truly
...
```

2 An Easy to Digest Example

If you were intimidated by technicalities of the above description, let's try another example that might be easier to digest.

I will start by telling a simple story of writing, compiling and executing and a simple program. Then, I will explain how I used the `\bash` command to not only tell a story, but also to play it live: that is, authoring a simple C program, compiling it and executing it, all from within LaTeX.

2.1 A "Hello, World" Program

2.1.1 Authoring

Let's first write a simple Hello, World! program in the C programming language:

```
% rm -f hello.c; cat << EOF > hello.c
/*
```

```
** hello.c: My first C program; it prints
** "Hello, World!", and dies.
*/

#include <stdio.h>

int main()
{
  printf("Hello, World!\n");
  return 0;
}
EOF
```

2.1.2 Compiling

Now, let's compile this program:

```
% cc hello.c
```

2.1.3 Executing

Finally, we can execute this program, and see that indeed, it prints the "Hello, World!" string.

```
% ./a.out
Hello, World!
```

2.2 Behind the Scenes

2.2.1 Authoring

What I wrote in the input to produce the `hello.c` program was:

```
\bash[script]
rm -f hello.c; cat << EOF > hello.c
/*
** hello.c: My first C program; it prints
** "Hello, World!", and dies.
*/

#include <stdio.h>

int main()
{
  printf("Hello, World!\n");
  return 0;
}
EOF
\END
```

In doing so, all the text between the `\bash` and `\END` was sent to a temporary file, which was then sent for execution. The `script` flag instructed `\bash` to list this file in the main document. This listing was prefixed with `%␣` to make it clear that it was input to `bash`.

2.2.2 Compiling

Next, I wrote

```
\bash[script,stdout]
cc hello.c
\END
```

As before, in doing that, I achieved two objectives: first, when LaTeX processed the input, it also invokes the C compiler to compile file `hello.c`, the file which I just created.

Second, thanks to the `script` flag, the command for compiling this program was included in the printed version of this document. The `stdout` option instructed `\bash` to include plain messages, i.e., not error messages, produced by the compiler in the printed version of this document. In this case, no such messages were produced.

2.2.3 Executing

Finally, I wrote

```
\bash[script,stdout]
./a.out
\END
```

to run the program I just wrote. The `stdout` adds to my listing the output that this execution produces, i.e., the string `Hello, World!` that this execution produces to the standard output.

3 Dealing with Shell Command Errors

Using bashful to demonstrate my *Hello, World!* program, made sure that the story I told is accurate: I really did everything I said I did. More accurately, the `\bash` command acted as my proxy, and did it for me.

Luckily, my `hello.c` program was correct. But, if it was not, the `\bash` command would have detected the error, and would have stopped the LaTeX process, indicating that the compilation did not succeed. More specifically, the `\bash` command

1. collects all commands up to `\END`;
2. places these commands in a script file;
3. change directory to a designated directory if the `hide` option is set (the `dir` option sets the directory name);
4. executes this script file, redirecting its standard output and its standard error streams to distinct files;
5. checks whether the exit code of the execution indicates an error (i.e., exit code which is different from 0), and if so, place this exit code in a distinct file;
6. checks whether the file containing the standard error is empty, and if not, pauses execution after displaying an error message;
7. checks whether the file containing the exit code is empty, and if not, pauses execution after displaying an error message;
8. lists, if requested to, the script file;
9. lists, if requested to, the file containing the standard output; and,
10. lists, if requested to, the file containing the standard error;

Let me demonstrate a situation in which the execution of the script generates an error. To do that, I will write a short LaTeX file, named `minimal.tex` which tries to use `\bash` to compile an incorrect C program. Since `minimal.tex` contains `\END`, I will have to author this file in three steps:

1. Creating the header of `minimal.tex`:

```
% cat << EOF > minimal.tex
\documentclass{standalone}
\usepackage{bashful}
\begin{document}
This document creates a simple erroneous C program
  and then compiles it:
\bash[script,stdout]
echo "main(){return int;}" > error.c; cc error.c
EOF
```

2. Adding `\END` to `minimal.tex`

```
% echo "\\END" >> minimal.tex
```

3. Finalizing `minimal.tex`

```
% echo "\\end{document}" >> minimal.tex
```

Let me now make sure `minimal.tex` was what I expect it to be:

```
% cat minimal.tex
\documentclass{standalone}
\usepackage{bashful}
\begin{document}
This document creates a simple erroneous C program
  and then compiles it:
\bash[script,stdout]
echo "main(){return int;}" > error.c; cc error.c
\END
\end{document}
```

I am now ready to run `minimal.tex` through LaTeX, but since I will not run the `latex` command myself, I will make sure a `q` character is sent to it when the anticipated error occurs.

```
% yes q | pdflatex -shell-esc minimal.tex
| fmt -s -w 55
This is pdfTeX, Version 3.1415926-1.40.11 (TeX
Live 2010)
 \write18 enabled.
entering extended mode
(./minimal.tex
LaTeX2e <2009/09/24>
Babel <v3.81> and hyphenation patterns for english,
dumylang, nohyphenation, ge
rman-x-2009-06-19, ngerman-x-2009-06-19, afrikaans,
ancientgreek, ibycus, arabi
c, armenian, basque, bulgarian, catalan, pinyin,
coptic, croatian, czech, danis
h, dutch, ukenglish, usenglishmax, esperanto,
estonian, ethiopic, farsi, finnis
h, french, galician, german, ngerman, swissgerman,
monogreek, greek, hungarian,
 icelandic, assamese, bengali, gujarati, hindi,
 kannada, malayalam, marathi, or
iya, panjabi, tamil, telugu, indonesian, interlingua,
irish, italian, kurmanji,
 lao, latin, latvian, lithuanian, mongolian,
 mongolianlmc, bokmal, nynorsk, pol
ish, portuguese, romanian, russian, sanskrit, serbian,
slovak, slovenian, spani
sh, swedish, turkish, turkmen, ukrainian,
uppersorbian, welsh, loaded.
(/usr/local/texlive/2010/texmf-dist/tex/latex/standalone/standalone.cls
Document Class: standalone 2010/02/28 v0.4 Class to
compile TeX sub-files stand
alone
(/usr/local/texlive/2010/texmf-dist/tex/latex/oberdiek/kvoptions.sty
(/usr/local/texlive/2010/texmf-dist/tex/latex/graphics/keyval.sty)
(/usr/local/texlive/2010/texmf-dist/tex/generic/oberdiek/kvsetkeys.sty
(/usr/local/texlive/2010/texmf-dist/tex/generic/oberdiek/infwarerr.sty)
(/usr/local/texlive/2010/texmf-dist/tex/generic/oberdiek/etexcmds.sty)))
(/usr/local/texlive/2010/texmf-dist/tex/latex/base/article.cls
Document Class: article 2007/10/19 v1.4h Standard
LaTeX document class
(/usr/local/texlive/2010/texmf-dist/tex/latex/base/size10.clo))
(/usr/local/texlive/2010/texmf-dist/tex/latex/standalone/standalone.cfg)
(/usr/local/texlive/2010/texmf-dist/tex/latex/preview/preview.sty
(/usr/local/texlive/2010/texmf-dist/tex/latex/preview/prtightpage.def)))
(./bashful.sty
(/usr/local/texlive/2010/texmf-dist/tex/generic/oberdiek/catchfile.sty
(/usr/local/texlive/2010/texmf-dist/tex/generic/oberdiek/ltxcmds.sty))
(/usr/local/texlive/2010/texmf-dist/tex/latex/xkeyval/xkeyval.sty
(/usr/local/texlive/2010/texmf-dist/tex/generic/xkeyval/xkeyval.tex))
(/usr/local/texlive/2010/texmf-dist/tex/latex/listings/listings.sty
(/usr/local/texlive/2010/texmf-dist/tex/latex/listings/lstmisc.sty)
(/usr/local/texlive/2010/texmf-dist/tex/latex/listings/listings.cfg)))
(./minimal.aux)
Preview: Fontsize 10pt
Preview: PDFoutput 1
(./minimal.sh) (./minimal.stdout)
Preview: Tightpage -32891 -32891 32891 32891
[1{/usr/local/texlive/2010/texmf-var/fonts/map/pdftex/updmap/pdftex.map}]
(./minimal.aux)
)</usr/local/texlive/2010/texmf-dist/fonts/type1/public/amsfont
```

You can see that when LaTeX tried to process `minimal.tex`, it stopped execution while indicating that file `minimal.stderr` was not empty after the compilation. The first line of `minimal.stderr` was displayed, and I was advised to examine this file myself. Inspecting `minimal.stderr`, we see the C compiler error messages:

```
% cat minimal.stderr
error.c: In function   m a i n   :
error.c:1: error: expected expression before   i n t
```

Note that the failure to compile `hello.c`, did not stop `\bash` from including this file in the source.

Here is what `minimal.pdf` looks like:

> This document creates a simple erroneous C program and then compiles it:
>
> ```
> % echo "main(){return int;}" > error.c; cc error.c
> ```

4 Customization

4.1 Package Options

Options to the `\bashful` package passed using the xkeyval syntax:

`hide` = ⟨`true`/`false`⟩ `false`
If `true`, scripts are executed in a designated directory; if `false`, scrips are executed in the current working directory.

`dir` = ⟨*directoryName*⟩
If `hide` option is `true`, then scripts are executed in this directory. Initial value of this options is `_00`. Note that if you use TeXlive 2010, you have to configure certain security flags to make it possible to write to directories whose name start with a dot, or to directories which are not below the current working directory.

`verbose` = ⟨`true`/`false`⟩ `false`
If `true`, be chatty.

4.2 Command Options

Options to `\bash` command are passed using the xkeyval syntax:

4.2.1 File names

`scriptFile` = ⟨*fileName*⟩ `\jobname.sh`
Name of file into which the script instructions are spilled prior to execution. The default is `\jobname.sh`; this file will be reused by all `\bash` commands in your documents. This is rarely a problem, since these scripts execute sequentially.

`stdoutFile` = ⟨*fileName*⟩ `\jobname.stdout`
Name of file into which the shell standard output stream is redirected.

`stderrFile` = ⟨*fileName*⟩ `\jobname.stderr`
Name of file into which the shell standard error stream is redirected.

`exitCodeFile` = ⟨*fileName*⟩ `\jobname.stderr`
Name of file into which the shell standard error stream is redirected.

4.2.2 Listing Structure

`script` = ⟨`true`/`false`⟩ `false`
If `true`, the content of `scriptFile` is listed in the main document.

`stdout` = ⟨`true`/`false`⟩ `false`
If `true`, the content of `stdoutFile` is listed in the main document. If both `script` and `stdout` are `true`, then `scriptFile` is listed first, and leaving no vertical space, `stdoutFile` is listed next.

`stderr` = ⟨`true`/`false`⟩ `false`
If `true`, the content of `stderrFile` is listed in the main document, following `scriptFile` (if `script` is `true`) and `stdoutFile` (if `stdout` is `true`).

4.2.3 Tolerance to Errors

`ignoreExitCode` = ⟨`true`/`false`⟩ `false`
When `true` `\bash` will consider an execution correct even if its exit code is not 0.

`ignoreStderr` = ⟨`true`/`false`⟩ `false`
When `true` `\bash` will consider an execution correct even if produces output to the standard error stream.

4.2.4 Appearance

`prefix` = ⟨tokens⟩ `%␣`
String that prefixes the listing of `scriptFile`.

`environment` = ⟨enrionmentName⟩ `none`
Name of LaTeX environment (e.g., `quote`) in which the listing is wrapped.

4.2.5 Miscellaneous

`verbose` = ⟨`true`/`false`⟩ `false`
If `true`, the package logs every step it takes.

4.3 Listings Styles

Package listing is used for all listing both the executed shell commands and their output.

4.3.1 Listings Style for Script File

Style `bashfulScript` is used for displaying the executed shell commands (when option `script` is used). The current definition of this style is:

```
\lstdefinestyle{bashfulScript}{
  basicstyle=\ttfamily,
  keywords={},
  showstringspaces=false}
```

Redefine this style to match your needs.

4.3.2 Listings Style for Standard Output

Style `bashfulStdout` is used for displaying the output of the executed shell commands (when option `stdout` is used). The current definition is:

```
% listings style for the stdoutFile, can be redefined by client
\lstdefinestyle{bashfulStdout}{
  basicstyle=\sl\ttfamily,
  keywords={},
  showstringspaces=false
}%
```

Redefine this style to match your needs.

Style `bashfulStderr` is used for displaying the output of the executed shell commands (when option `stderr` is used).

```
\lstdefinestyle{bashfulStderr}{
  basicstyle=\sl\ttfamily\color{red},
  keywords={},
  showstringspaces=false
}
```

Redefine this style to match your needs.

5 Interaction with Other Packages

This packages tries to work around a bug in `polyglossia` by which `\texttt` is garbled upon switching to languages which do not use the Latin alphabet. Also, in case bidirectional TEXing is in effect, bashful forces the listing to be left-to-right.

6 History

Version 0.91 Initial release.

Version 0.92
- Added `ignoreExitCode`, `ignoreStderr`, `stderr`, `exitCodeFile` command options.
- Renamed `list` to `script`.
- Added `hide` and `dir` package options.

7 Future

The following may get implemented some day.

1. *Package options.* Currently all options are passed to the command itself.
2. *A `clean` option.* This option will automatically erase files generated for storing the script, and its standard output and standard error streams.
3. *A `noclobber` option.* This option will make this package safer, by reducing the risk of accidentally erasing existing files.

8 Acknowledgments

The manner by which `\bash` collects its arguments is based on that of tobiShell. Martin Scharrer tips on TEX internals were invaluable.

A Source of `bashful.sty`

```
% Copyright (C) 2011 by Yossi Gil  yogi@cs.technion.ac.il
% ------------------------------------------------------------------------
% This work may be distributed and/or modified under the conditions of the
% LaTeX Project Public License (LPPL), either version 1.3 of this license or
% (at your option) any later version.  The latest version of this license is in
% http://www.latex-project.org/lppl.txt and version 1.3 or later is part of all
% distributions of LaTeX version 2005/12/01 or later.
%
% This work has the LPPL maintenance status ‘maintained’.
%
```

```
% The Current Maintainer of this work is Yossi Gil.
%
% This work consists of the files bashful.tex and bashful.sty and the derived
% bashful.pdf

\NeedsTeXFormat{LaTeX2e}%

% Auxiliary identification information
\newcommand\date@bashful{2011/03/12}%
\newcommand\version@bashful{V 0.92}%
\newcommand\author@bashful{Yossi Gil}%
\newcommand\mail@bashful{yogi@cs.technion.ac.il}%
\newcommand\signature@bashful{%
  bashful \version@bashful{} by
  \author@bashful{} \mail@bashful
}%

% Identify this package
\ProvidesPackage{bashful}[\date@bashful{} \signature@bashful:
  Write and execute a bash script within LaTeX, with, or
  without displaying the script and/or its output.
]
\PackageInfo{bashful}{This is bashful, \signature@bashful}%

\RequirePackage{catchfile}
% Use xkeyval for retrieving parameters
\RequirePackage{xkeyval}%

% If true, all activities take place in a designated directory.
\newif\if@hide@BL@\@hide@BL@false

% This is the default name for a directory in which processing should
% take place if \@hide@BL@true.
\def\directory@BL{_00}

% Use listing to display bash scripts.
\RequirePackage{listings}%

  % listings style for the script, can be redefined by client
  \lstdefinestyle{bashfulScript}{
    basicstyle=\ttfamily,
    keywords={},
    showstringspaces=false}%
  % listings style for the stdoutFile, can be redefined by client
  \lstdefinestyle{bashfulStdout}{
    basicstyle=\sl\ttfamily,
    keywords={},
    showstringspaces=false
  }%
  % listings style for the stderrFile, can be redefined by client
  \lstdefinestyle{bashfulStderr}{
    basicstyle=\sl\ttfamily\color{red},
    keywords={},
    showstringspaces=false
  }%


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % File Name keys in alphabetical order:
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

   % scriptFile: String = \BL@exitCodeFile: Where should the exit code be stored
   % if it is not zero.
  \edef\BL@exitCodeFile{\jobname.exitCode}%
  \define@cmdkey{bashful}[BL@]{exitCodeFile}{}%

  % scriptFile: String = \BL@scriptFile: Where should the script be saved?
  \edef\BL@scriptFile{\jobname.sh}%
```

```
\define@cmdkey{bashful}[BL@]{scriptFile}{}%

 % stderrFile: String = \BL@stderrFile: Where should the standard error
 % sream be saved?
\edef\BL@stderrFile{\jobname.stderr}%
\define@cmdkey{bashful}[BL@]{stderrFile}{}%

 % stdoutFile: String = \BL@stdoutFile: Where should the standard output stream
 % be saved?
\edef\BL@stdoutFile{\jobname.stdout}%
\define@cmdkey{bashful}[BL@]{stdoutFile}{}%

 % dir: String = \directory@BL: name of directory in which execution is going to take
 % place
\define@cmdkey{bashful}[BL@]{dir}{\def\directory@BL{#1}}%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% List configuration boolean keys
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % list: Boolean = \ifBL@script: Should we list the script we generate?
\define@boolkey{bashful}[BL@]{script}[true]{}%

 % stdout: Boolean = \ifBL@stderr: Should we list the standard error?
 \define@boolkey{bashful}[BL@]{stderr}[true]{}%

% stdout: Boolean = \ifBL@stdout: Should we list the standard output?
\define@boolkey{bashful}[BL@]{stdout}[true]{}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Error checking Boolean keys.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% stdout: Boolean = \ifBL@ignoreExitCode: Should we ignore the exit
% code?
\define@boolkey{bashful}[BL@]{ignoreExitCode}[true]{}

% stdout: Boolean = \ifBL@ignoreStderr: Should we ignore the exit
% code?
\define@boolkey{bashful}[BL@]{ignoreStderr}[true]{}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Miscelaneous keys
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% environment: String = \BL@environment: Which environment should we wrap
% the listings
\def\BL@environment{none@BL}%
\define@cmdkey{bashful}[BL@]{environment}{}%
\newenvironment{none@BL}{}{} % Default, empty environment for wrapping the listings

% prefix: String = \BL@prefix: What prefix should be printed before a listing.
\def\BL@prefix{\@percentchar\space}%
\define@cmdkey{bashful}[BL@]{prefix}{}%

 % shell: String = \BL@shell: Which shell should be used for execution?
\def\BL@shell{bash}%
\define@cmdkey{bashful}[BL@]{shell}{}%

% verbose: Boolean = \ifBL@verbose: Log every step we do
\define@boolkey{bashful}[BL@]{verbose}[true]{}%


\DeclareOptionX{hide}{\@hide@BL@true}
\DeclareOptionX{dir}{\@hide@BL@true\def\directory@BL{#1}}
\DeclareOptionX{verbose}{\BL@verbosetrue}
\ExecuteOptionsX{}
\ProcessOptionsX\relax
```

```
% \bash: the main command we define. It chains to \bashI which chains to
% \bashII, etc.
\begingroup
  %\where@BL
  \catcode`\^^M\active%
  \gdef\bash{%
    \logBL{Beginning a group so that all cat code changes are local}%
    \begingroup%
    \logBL{Making \^\^M a true newline}%
    \catcode`\^^M\active%
    \def^^M{^^J}%
    \logBL{Checking for optional arguments}%
    \@ifnextchar[\bashI{\bashI[]}%
  }%
\endgroup

% \bashI: Process the optional arguments and continue
\def\bashI[#1]{\setKeys@BL{#1}\bashII}

% \bashII: Set category codes of all characters to special, and proceed.
\begingroup
  \catcode`\^^M\active%
  \gdef\bashII{%
    \logBL{bashII: Making \^\^M a true new line}%
    \catcode`\^^M\active%
    \def^^M{^^J}%
    \logBL{bashII: Making all characters other}%
    \let\do\@makeother%
    \dospecials%
    \bashIII}%
\endgroup

% \bashIII: Consume all tokens until \END (but ignoring the preceding and
% terminating newline), and proceed.
\begingroup
  \catcode`\@=0\relax
  \catcode`\^^M\active
  @catcode`@\=12@relax%
  @gdef@bashIII^^M#1^^M\END{@bashIV{#1}@bashV{#1}@logBL{bashV: Done!}@endgroup}@endgroup
%

\newcommand\logBL[1]{\ifBL@verbose\typeout{L\the\inputlineno: #1}\fi}

% \bashIV: Process the tokens by storing them in a script file, and executing
% this file,
\newcommand\bashIV[1]{%
  \logBL{BashIV: begin}%
  \makeDirectory@BL
  \generateScriptFile@BL{#1}\relax
  \executeScriptFile@BL
  \logBL{BashIV: done}%
}%
\def\makeDirectory@BL{%
  \if@hide@BL@
    \logBL{Making directory \directory@BL}%
    \immediate\write18{mkdir -p \directory@BL}%
  \else
    \logBL{Using current directory}%
  \fi
}

\newcommand\splice[1]{%
  \bashIV{#1}%
  \expandFileName@BL{\BL@stdoutFile}%
  \CatchFileDef{\BL@file@contents}{\BL@stdoutFile}{\relax}%
  \ignorespaces\BL@file@contents\unskip
}
```

```
% listing the script file if required, and presenting the standard output and
% standar error if required.
\newcommand\bashV[1]{%
  \logBL{Wrapping up after execution}%
  \storeToFile@BL{\BL@prefix#1}{\BL@scriptFile}%
  \expandFileName@BL\BL@scriptFile
  \expandFileName@BL\BL@stdoutFile
  \expandFileName@BL\BL@stderrFile
  \logBL{Files are: \BL@scriptFile, \BL@stdoutFile, and \BL@stderrFile}%
  %\checkScriptErrors@BL
  \listEverything@BL
  \logBL{Wrap up done}}

\def\expandFileName@BL#1{%
  \logBL{expandFileName}%
  \logBL{Setting, if necessary, correct path of \noexpand#1 }%
  \if@hide@BL@
    \logBL{Prepending path (\directory@BL) to #1}%
    \edef#1{\directory@BL/#1}%
    \logBL{Obtained #1}%
  \fi
}

\def\setKeys@BL#1{%
  \logBL{Processing key=val pairs in options string [#1]}\relax
  \setkeys{bashful}{#1}%
}%

% Store the list of tokens in the first argument into our script file
\newcommand\generateScriptFile@BL[1]{%
  \storeToFile@BL{#1}{\BL@scriptFile}%
}%

\newwrite\writer@BL
% Store the list of tokens in the first argument into the file given
% in the second argument; prepend directory if necessary
\newcommand\storeToFile@BL[2]{%
  \logBL{  #2 :=^^J#1^^J}%
  \if@hide@BL@
     \logBL{File #2 will be created in \directory@BL}%
     \storeToFileI@BL{#1}{\directory@BL/#2}
  \else
     \logBL{File #2 will be created in current directory}%
     \storeToFileI@BL{#1}{#2}%
  \fi
  \logBL{Writing done!}%
}%

% Store the list of tokens in the first argument into the file given
% in the second argument; the second argument could be qualified with
% a directory name.
\newcommand\storeToFileI@BL[2]{%
  \logBL{Writing to file #2...}%
  \immediate\openout\writer@BL#2%
  \immediate\write\writer@BL{#1}%
  \immediate\closeout\writer@BL
}%

% Execute the content of our script file.
\newcommand\executeScriptFile@BL{%
  \edef\command@BL{\BL@shell \space \BL@scriptFile}%
  \if@hide@BL@
    \logBL{Adding a "cd command"}%
    \edef\command@BL{cd \directory@BL;\command@BL}
  \fi%
  \edef\command@BL{\command@BL \space >\BL@stdoutFile \space 2>\BL@stderrFile}%
  \edef\command@BL{\command@BL \space || echo $? >\BL@exitCodeFile}%
```

```
  \edef\command@BL{\BL@shell\space -c "\command@BL"}%
  \logBL{Executing:^^J \command@BL}%
  \immediate\write18{\command@BL}%
}%

\newread\reader@BL

% Issue an error message if errors found during execution
\newcommand\checkScriptErrors@BL{%
  \logBL{Checking for script errors}%
  \begingroup
  \newif\ifErrorsFound@\ErrorsFound@false
  \checkExitCodeFile@BL
  \ifdefined\exitCode@BL
    \logBL{Non zero exit code found (\exitCode@BL), and I was not instructed to ignore it}
    \ErrorsFound@true
  \fi
  \def\eoln{\par}
  \def\firstErrorLine{\par}
  \checkStderrFile@BL
  \logBL{I will print content of \BL@stderrFile\space (if found)}
  \ifx\firstErrorLine\eoln
    \relax
  \else
    \logBL{Standard error was not empty, and I was not instructed to ignore it}
    \message{Standard error not empty. Here is how
     ^^Jfile \BL@stderrFile\space begins:
     ^^J>>>>\space
     ^^Jbut, you really ought to examine this file yourself!}
    \ErrorsFound@true
  \fi
  \ifErrorsFound@
    \logBL{Issuing an error message since \BL@stderrFile\space was not empty}%
    \errmessage{Your shell script failed....}
  \else
    \logBL{Proceeding as usual}%
  \fi
  \endgroup
}%

\newcommand\checkExitCodeFile@BL{%
  \ifBL@ignoreExitCode
     \logBL{Ignoring \BL@exitCodeFile, as per command flag}%
  \else
    \logBL{Opening \BL@exitCodeFile}%
    \openin\reader@BL=\BL@exitCodeFile
    \ifeof\reader@BL
      \logBL{File \BL@exitCodeFile\space missing, exit code was probably 0}
    \else
      \logBL{File \BL@exitCodeFile\space exists, let's get the exit code}%
      \logBL{Reading first line of \BL@exitCodeFile}%
      \catcode`\^^M=5
      \read\reader@BL to \exitCode@BL
      \closein\reader@BL
    \fi
  \fi
}

\newcommand\checkStderrFile@BL{%
  \ifBL@stderr
    \logBL{Will be listing \BL@stderrFile, so erroneous content is ignored}%
  \else
    \ifBL@ignoreStderr
      \logBL{Ignoring \BL@stderrFile, as per command flag}%
    \else
      \checkStderrFileI@BL
    \fi
  \fi
```

```
}

\newcommand\checkStderrFileI@BL{%
  \logBL{Opening \BL@stderrFile}%
  \openin\reader@BL=\BL@stderrFile\relax
  \ifeof\reader@BL
    \BL@verbosetrue
    \logBL{Hmm... \BL@stderrFile\space does not exist (probably a package bug)}%
    \logBL{Switching to verbose mode}%
  \else
    \logBL{Reading first line of \BL@stderrFile}%
    \catcode`\^^M=5
    \read\reader@BL to \firstErrorLine
    \ifeof\reader@BL
      \ifx\firstErrorLine\eoln
        \logBL{File \BL@stderrFile\space is empty}
      \else
        \logBL{File \BL@stderrFile\space has one line [\firstErrorLine]}%
        \ErrorsFound@true
      \fi
    \else
     \logBL{File \BL@stderrFile\space has two lines or more}%
     \ErrorsFound@true
   \fi
   \closein\reader@BL
}

% List the contents of the script, stdout and stderr, as per the flags.
\newcommand\listEverything@BL{%
  \logBL{Checking whether any listings are required}%
  \newif\if@listSomething@BL@
  \ifBL@script\@listSomething@BL@true\fi
  \ifBL@stdout\@listSomething@BL@true\fi
  \ifBL@stderr\@listSomething@BL@true\fi
  \if@listSomething@BL@
    \beginWrappingEnvironment@BL
    \listEverythingI@BL
    \endWrappingEnvironment@BL
  \else
    \logBL{Nothing has to be listed}%
  \fi
}

% Auxiliary macro to list the contents of the script, stdout and stderr, as per
% the flags.
\newcommand\listEverythingI@BL{%
  \logBL{Laying out the correct \noexpand\lstinputlisting commands}%1
  \ifBL@script\listScript@BL\BL@scriptFile\fi
  \ifBL@stdout\listStdout@BL\BL@stdoutFile\fi
  \ifBL@stderr\listStderr@BL\BL@stderrFile\fi
}%

\newcommand\listScript@BL[1]{%
  \logBL{Listing script: #1}
  \def\flags@BL{style=bashfulScript}
  \logBL{Initial flags for listing #1 are \flags@BL}
  \ifBL@stdout\edef\flags@BL{\flags@BL, belowskip=0pt}\fi
  \ifBL@stderr\edef\flags@BL{\flags@BL, belowskip=0pt}\fi
  \doList@BL#1\flags@BL
}

\newcommand\listStdout@BL[1]{%
  \logBL{Listing stdout: #1}
  \edef\flags@BL{style=bashfulStdout}
  \logBL{Initial flags for listing stdout file are \flags@BL}
  \ifBL@script\edef\flags@BL{\flags@BL, aboveskip=0pt}\fi
  \ifBL@stderr\edef\flags@BL{\flags@BL, belowskip=0pt}\fi
  \doList@BL#1\flags@BL
```

```
}%

\newcommand\listStderr@BL[1]{%
  \logBL{Listing stderr: #1}%
  \def\flags@BL{style=bashfulStderr}%
  \logBL{Initial flags for listing stderr file are \flags@BL}
  \ifBL@script\edef\flags@BL{\flags@BL, aboveskip=0pt}\fi
  \ifBL@stdout\edef\flags@BL{\flags@BL, aboveskip=0pt}\fi
  \doList@BL#1\flags@BL
}%

\newcommand\doList@BL[2]{%
    \logBL{Flags for listing #1 are #2}%
    \expandafter\lstset\expandafter{#2}%
    \lstinputlisting{#1}%
  }%

\def\beginWrappingEnvironment@BL{%
  \logBL{Beginning environment \BL@environment}%
  \expandafter\csname\BL@environment\endcsname
  \forceLTR@BL
  \fixPolyglossiaBug@BL
}%

\def\endWrappingEnvironment@BL{%
  \expandafter\csname end\BL@environment\endcsname
}%

\newcommand\fixPolyglossiaBug@BL{%
  \logBL{Trying to fix a Polyglossia package bug}%
  \ifdefined\ttfamilylatin
    \logBL{Replacing \noexpand\ttfamily with \noexpand\ttfamilylatin}%
    \let\ttfamily=\ttfamilylatin
    \logBL{Replacing \noexpand\rmfamily with \noexpand\rmfamilylatin}%
    \let\rmfamily=\rmfamilylatin
    \logBL{Replacing \noexpand\sffamily with \noexpand\sffamilylatin}%
    \let\sffamily=\sffamilylatin
    \logBL{Replacing \noexpand\normalfont with \noexpand\normalfontlatin}%
    \let\normalfont=\normalfontlatin
  \else
    \logBL{Polyglossia package probably not loaded}%
    \relax
 \fi
}%

\newcommand\forceLTR@BL{%
  \logBL{Making sure we are not in right-to-left mode}%
  \ifdefined\setLTR
    \logBL{Command \noexpand\setLTR is defined, invoking it}%
    \setLTR
  \else
    \logBL{Command \noexpand\setLTR is not defined, we are probably LTR}%
    \relax
  \fi
}%
```

B Source of `bashful.tex`

```
\documentclass{ltxdoc} % Process with xelatex -shell-escape
\usepackage[verbose]{bashful}

\usepackage[colorlinks=true]{hyperref}
\usepackage{gensymb}
\usepackage{graphicx}
\usepackage{metalogo}
\usepackage{xkvview}
\usepackage{xspace}
```

```
\usepackage{amsmath}
\usepackage{multicol}

\newcommand\me{bashful}
\newcommand\bashful{\textsf{\me}\xspace}
  \lstdefinestyle{input}{
    basicstyle=\ttfamily,
    showstringspaces=false,
    aboveskip=0pt,
    belowskip=0pt}%

\title{The \bashful Package\thanks{
   Copyright \copyright{} 2011 by Yossi Gil
    \url{mailto:yogi@cs.technion.ac.il}.
  This work may be distributed and/or modified under the conditions of the
    \emph{\LaTeX{} Project Public License} (LPPL), either version 1.3 of this
    license or (at your option) any later version.
The latest version of this license is in
  \url{http://www.latex-project.org/lppl.txt} and version 1.3 or later
  is part of all distributions of \LaTeX{} version 2005/12/01 or later.
This work has the LPPL maintenance status `maintained'.
The Current Maintainer of this work is Yossi Gil.
This work consists of the files \texttt{\me.tex} and \texttt{\me.sty}
  and the derived file
  \texttt{\me.pdf}
}}

\author{Yossi Gil\thanks{\url{mailto:yogi@cs.Technion.ac.IL}}\\
   \normalsize Department of Computer Science\\
   \normalsize The Technion---Israel Institute of Technology\\
   \normalsize Technion City, Haifa 32000, Israel
}

\makeatletter
\date{\date@bashful\thanks{
      This document describes \bashful \version@bashful.}}
\makeatother

\begin{document}

\bash
cat << EOF > README
The bashful package, v 0.92

This package makes it possible to execute bash scripts from within LaTeX. The
main application is in writing computer-science texts, in which you want to
make sure the programs listed in the document are executed directly from the
input.

This package may be distributed and/or modified under the LaTeX Project Public
License, version 1.3 or higher (your choice). The latest version of this
license is at: http://www.latex-project.org/lppl.txt

This work is author-maintained (as per LPPL maintenance status)
by Yossi Gil, <yogi@cs.Technion.ac.i>
EOF
\END

\bash[verbose,stdoutFile=bashful.date]
stat -c %y bashful.sty | sed -e s+-+/+g -e 's/ .*//g' > date
\END

\maketitle


\begin{abstract}
\parindent 1.5ex
\parskip 0.5em
```

```
\sl
It is sometimes useful to ``\emph{escape-to-shell}'' from within
  \LaTeX{}.
The most obvious application is when the document
  explains something about the working of a computer program.
Your text would be more robust to changes, and easier to write,
  if all the examples it gives, are run directly from
  within \LaTeX{}.

To facilitate this and other applications,
  package \bashful{} provides a convenient interface to \TeX's
  primitive \verb+\write18+---the execution of shell commands from within
  your input files, also known as \emph{shell escape}.
Text between \verb+\bash+ and \verb+\END+ is executed by
   \href
        {http://en.wikipedia.org/wiki/Bash_%28Unix_shell%29}
        {\texttt{bash}},
  a popular Unix command line interpreter.
Various flags control whether the executed commands and their output
  show up in the printed document, and whether they are saved
  to files.

Although provisions are made for using shells other
  than \texttt{bash}, this package may \emph{not} operate without
  modifications on  Microsoft's operating systems.
\end{abstract}

\begin{multicols}{2}
\footnotesize
\tableofcontents
\end{multicols}

\parindent 1.5ex
\parskip 0.5em

\section{Introduction}

\bash[verbose,scriptFile=temperature.sh,stdoutFile=temperature.tex]
location=Jerusalem,Israel
server="http://www.Google.com/ig/api"
request="$server?weather=$location"
wget -q -O - $request |\
tr "<>" "\012\012" |\
grep temp_c |\
sed 's/[^0-9]//g'
\END

\bash[verbose,scriptFile=condition.sh,stdoutFile=condition.tex]
location=Jerusalem,Israel
server="http://www.Google.com/ig/api"
request="$server?weather=$location"
wget -q -O - $request |\
tr "<>" "\012\012" |\
grep "condition data" |\
head -n 1 |\
sed -e 's/^.*="//' -e 's/"\/*//' |\
tr 'A-Z' 'a-z'
\END

At the time I run this document through \LaTeX{},
  the temperature in Jerusalem, Israel,
  was~\emph{\input{temperature}\unskip\celsius},
  while the weather condition was
  \emph{\input{condition}}\unskip.

You may not care so much about these bits of truly
  ephemeral information,
```

```
  but you may be surprised that they were produced
  by the very process of \LaTeX{}ing the input.

How did I do that? Well, the first step was to write
  a series of shell commands that retrieve the current temperature,
  and another such series to obtain the current
  weather conditions.
This task required connection to
  \href{http://www.Google.com/support/forum/p/%
        apps-apis/thread?tid=0c95e45bd80def1a&hl=en}%
  {Google's weather service} and
  minimal dexterity with Unix pipes and filters to process the output.

My command series to obtain the current temperature was:

\begin{minipage}{\textwidth}
\begin{quote}
  \lstinputlisting[style=input]{temperature.sh}
\end{quote}
\end{minipage}

while the weather condition was obtained by

\begin{minipage}{\textwidth}
\begin{quote}
  \lstinputlisting[style=input]{condition.sh}
\end{quote}
\end{minipage}

The second step was coercing \LaTeX{} to run these commands
  while processing my document.
To do that, I used package \bashful,
\begin{verbatim}
\usepackage{bashful}
\end{verbatim}
And, then, I wrapped each of these two series within
  a \verb+\bash+\ldots\verb+\END+ pair.

The \verb+\bash+ command, offered by this package,
  takes all subsequent lines, stopping at  the closing \verb+\END+,
  places these in a file, and then
  lets the \texttt{bash} shell interpreter execute this file.

Allowing \LaTeX{} to run arbitrary shell commands can be
  dangerous---you never know whether that nice looking \texttt{.tex}
  file you received by email was prepared by a friend or
  a foe.
This is the reason that you have to tell \LaTeX{}
  explicitly that shell escapes
  are allowed.
The \texttt{-shell-esc} flag does that.
To process my document, I typed, at the command line,
\begin{quote}
  \tt
  \% latex -shell-escape \jobname.tex
\end{quote}

What I actually wrote in the input
  to produce the temperature in
  Jerusalem, Israel was:

\begin{minipage}{\textwidth}
\begin{quote}
\noindent\verb+\bash[verbose,scriptFile=temperature.sh,stdoutFile=temperature.tex]+
\lstinputlisting[style=input,belowskip=0pt]{temperature.sh}
\verb+\END+\\
\end{quote}
\end{minipage}
```

```
The flags passed to the \verb+bash+ control sequence above instructed it:
  \begin{enumerate}
    \item to be verbose, typing out a detailed log of everything it did;
    \item to save the shell commands in a script file named
          \texttt{temperature.sh}; and,
    \item to store the standard output of the script in a file named
          \texttt{temperature.tex}.
  \end{enumerate}

To obtain the current weather condition in the capital I wrote:

\begin{minipage}{\textwidth}
\begin{quote}
\noindent\verb+\bash[verbose,scriptFile=condition.sh,stdoutFile=condition.tex]+
\lstinputlisting[style=input]{condition.sh}
\verb+\END+
\end{quote}
\end{minipage}

I wrote these two just after my \verb+\begin{document}+.
When \LaTeX{} encountered these, it executed the bash commands and
  created two files \texttt{temperature.tex} and \texttt{condition.tex}.

Subsequently, I could use the content of these files by writing:


\begin{quote}
\bash
sed -n "/^At the time/,/^You may not/ p" bashful.tex  > init.tex
\END

\lstinputlisting[style=input,belowskip=0pt]{init.tex}\ldots
\end{quote}

\section{An Easy to Digest Example}
If you were intimidated by technicalities of the
  above description, let's try another example
  that might be easier to digest.

I will start by telling a simple story
  of writing, compiling and executing and
    a simple program.
Then, I will explain how I used the \verb+\bash+
  command to not only tell a story, but
  also to play it live: that is, authoring
  a simple~C program, compiling it and executing
  it, all from within \LaTeX{}.

\subsection{A ``Hello, World'' Program}

\subsubsection{Authoring}
Let's first write a simple
  \href{http://en.wikipedia.org/wiki/Hello_world_program}
    {Hello, World!} program in the
    \href{http://en.wikipedia.org/wiki/C_(programming_language)}
    {C programming language}:


\bash[verbose,environment=quote,script]
rm -f hello.c; cat << EOF > hello.c
/*
** hello.c: My first C program; it prints
** "Hello, World!", and dies.
*/

#include <stdio.h>
```

```
int main()
{
  printf("Hello, World!\n");
  return 0;
}
EOF
\END

\subsubsection{Compiling}
Now, let's compile this program:
\bash[environment=quote,script,stdout]
cc hello.c
\END

\subsubsection{Executing}
Finally, we can execute this program,
  and see that indeed, it prints the ``Hello, World!''
  string.
\bash[environment=quote,script,stdout]
./a.out
\END


\subsection{Behind the Scenes}
\subsubsection{Authoring}
What I wrote in the input to produce the
  \texttt{hello.c} program was:

\begin{minipage}{\textwidth}
\begin{quote}
\begin{verbatim}
\bash[script]
rm -f hello.c; cat << EOF > hello.c
/*
** hello.c: My first C program; it prints
** "Hello, World!", and dies.
*/

#include <stdio.h>

int main()
{
  printf("Hello, World!\n");
  return 0;
}
EOF
\END
\end{verbatim}
\end{quote}
\end{minipage}

In doing so, all the text between the \verb+\bash+
  and \verb+\END+ was sent to a temporary file,
    which was then sent for execution.
The \texttt{script} flag instructed \verb+\bash+
  to list this file in the main document.
This listing was prefixed with \verb*+% +
  to make it clear that it was input to \texttt{bash}.

\subsubsection{Compiling}
Next, I wrote
\begin{quote}
\begin{verbatim}
\bash[script,stdout]
cc hello.c
\END
\end{verbatim}
\end{quote}
```

```
350
     As before, in doing that, I achieved two objectives:
       first, when \LaTeX{} processed
       the input, it also invokes the~C compiler to compile
       file \texttt{hello.c}, the file which I just created.

     Second, thanks to the \texttt{script} flag,
       the command for compiling this program
       was included in the printed version of
       this document.
360  The \texttt{stdout} option instructed \verb+\bash+
      to include plain messages, i.e., not error messages,
      produced by the compiler in
       the printed version of this document.
     In this case, no such messages were produced.


     \subsubsection{Executing}

     Finally, I wrote
370  \begin{quote}
     \begin{verbatim}
     \bash[script,stdout]
     ./a.out
     \END
     \end{verbatim}
     \end{quote}
     to run the program I just wrote.
     The \texttt{stdout} adds to my listing
       the output that this execution produces, i.e.,
380    the string \texttt{Hello, World!} that this
       execution produces to the standard output.

     \section{Dealing with Shell Command Errors}
     Using \bashful{} to demonstrate
       my \emph{Hello, World!} program, made
       sure that the story I told is accurate:
     I really did everything I said I did.
     More accurately, the \verb+\bash+ command
       acted as my proxy, and did it for me.
390
     Luckily, my \texttt{hello.c} program was
       correct.
     But, if it was not, the \verb+\bash+ command would have detected
       the error, and would have stopped the \LaTeX{} process,
       indicating that the compilation did not succeed.
     More specifically, the \verb+\bash+ command
     \begin{enumerate}
     \item collects all commands up to \verb+\END+;
     \item places these commands in a script file;
400  \item change directory to a designated directory if the \texttt{hide}
         option is set (the \texttt{dir} option sets the directory name);
     \item executes this script file, redirecting its standard output
           and its standard error streams to distinct files;
     \item checks whether the exit code of the execution indicates an error
       (i.e., exit code which is different from~$0$), and if so,
         place this exit code in a distinct file;
     \item checks whether the file containing the standard error is empty,
         and if not, pauses execution after displaying an error message;
     \item checks whether the file containing the exit code is empty,
410      and if not, pauses execution after displaying an error message;
     \item lists, if requested to, the script file;
     \item lists, if requested to, the file containing the standard output; and,
     \item lists, if requested to, the file containing the standard error;
     \end{enumerate}

     Let me demonstrate a situation in which the execution of
       the script generates an error.
```

```
To do that, I will write a short \LaTeX{} file, named \texttt{minimal.tex}
  which tries to use \verb+\bash+ to compile an incorrect~C program.
Since \texttt{minimal.tex} contains \verb+\END+,
  I will have to author this file in three steps:
\begin{enumerate}
\item Creating the header of \texttt{minimal.tex}:
\bash[script]
cat << EOF > minimal.tex
\documentclass{standalone}
\usepackage{bashful}
\begin{document}
This document creates a simple erroneous C program
  and then compiles it:
\bash[script,stdout]
echo "main(){return int;}" > error.c; cc error.c
EOF
\END
\item Adding \verb+\END+ to \texttt{minimal.tex}
\bash[script]
echo "\\END" >> minimal.tex
\END
\item Finalizing \texttt{minimal.tex}
\bash[script]
echo "\\end{document}" >> minimal.tex
\END
\end{enumerate}

Let me now make sure \texttt{minimal.tex} was what I expect it to be:

\begin{minipage}{\textwidth}
\bash[script,stdout]
cat minimal.tex
\END
\end{minipage}

I am now ready to run \texttt{minimal.tex} through \LaTeX{},
  but since I will not run the \texttt{latex} command myself,
  I will make sure a \texttt{q} character is sent to it
  when the anticipated error occurs.

\begin{minipage}{\textwidth}
  \lstdefinestyle{bashfulStdout}{
    showstringspaces=false,
    basicstyle=\small\ttfamily,
  }%
\bash[script,stdout]
yes q | pdflatex -shell-esc minimal.tex  | fmt -s -w 55
\END
\end{minipage}

You can see that when \LaTeX{} tried to process \texttt{minimal.tex},
  it stopped execution while indicating that file
  \texttt{minimal.stderr} was not
  empty after the compilation. The first line of \texttt{minimal.stderr}
  was displayed, and I was advised to examine this file myself.
Inspecting \texttt{minimal.stderr}, we see the C compiler error messages:

\begin{minipage}{\textwidth}
\bash[script,stdout]
cat minimal.stderr
\END
\end{minipage}

Note that the failure to compile \texttt{hello.c},
  did not stop \verb+\bash+ from including
  this file in the source.

Here is what \texttt{minimal.pdf} looks like:
```

```
\begin{center}
  \includegraphics[scale=0.9]{minimal.pdf}
\end{center}

\section{Customization}

\newcommand\option[3]{%
      \noindent\(
          \text{\bfseries\texttt{#1}}
          =
          \langle\text{{#2}}\rangle
      \)
      \hfill\texttt{#3}\\}
\subsection{Package Options}
Options to the \verb+\bashful+ package passed using the \textsf{xkeyval} syntax:

\option{hide}{\texttt{true}/\texttt{false}}{\texttt{false}}
If \texttt{true}, scripts are
  executed in a designated directory;
   if \texttt{false}, scrips are executed
   in the current working directory.

\option{dir}{\sl directoryName}{}
If \texttt{hide} option is \texttt{true}, then
  scripts are executed in this directory.
Initial value of this options is \verb+_00+.
Note that if you use \TeX{}live 2010, you have to configure certain
  security flags to make it possible to write to directories
  whose name start with a dot, or to directories
  which are not below the current working directory.

\option{verbose}{\texttt{true}/\texttt{false}}{\texttt{false}}
If \texttt{true}, be chatty.

\subsection{Command Options}

Options to \verb+\bash+ command
  are passed using the \textsf{xkeyval} syntax:



\subsubsection{File names}
\option{scriptFile}{\sl fileName}{\textbackslash jobname.sh}
Name of file into which the script instructions are spilled prior
  to execution.
The default is \verb+\jobname.sh+; this file
  will be reused by all \verb+\bash+ commands in your documents.
  This is rarely a problem, since these scripts
  execute sequentially.

\option{stdoutFile}{\sl fileName}{\textbackslash jobname.stdout}
Name of file into which the shell standard output stream is
  redirected.


\option{stderrFile}{\sl fileName}{\textbackslash jobname.stderr}
Name of file into which the shell standard error stream is
  redirected.

\option{exitCodeFile}{\sl fileName}{\textbackslash jobname.stderr}
Name of file into which the shell standard error stream is
  redirected.

\subsubsection{Listing Structure}
\option{script}{\texttt{true}/\texttt{false}}{\texttt{false}}
```

```
If \texttt{true}, the content of \texttt{scriptFile}
  is listed in the main document.

\option{stdout}{\texttt{true}/\texttt{false}}{\texttt{false}}
If \texttt{true}, the content of \texttt{stdoutFile}
  is listed in the main document.
If both \texttt{script} and \texttt{stdout} are
  \texttt{true}, then \texttt{scriptFile} is listed
  first, and leaving no vertical space,
  \texttt{stdoutFile} is listed next.

\option{stderr}{\texttt{true}/\texttt{false}}{\texttt{false}}
If \texttt{true}, the content of \texttt{stderrFile}
  is listed in the main document, following
  \texttt{scriptFile} (if \texttt{script} is
  \texttt{true})
  and
  \texttt{stdoutFile} (if \texttt{stdout} is
  \texttt{true}).

\subsubsection{Tolerance to Errors}
\option{ignoreExitCode}{\texttt{true}/\texttt{false}}{\texttt{false}}
When
  \texttt{true} \verb+\bash+ will consider
    an execution correct even if its exit code
    is not 0.

\option{ignoreStderr}{\texttt{true}/\texttt{false}}{\texttt{false}}
 When \texttt{true} \verb+\bash+ will consider
    an execution correct even if produces
    output to the standard error stream.

\subsubsection{Appearance}

\option{prefix}{tokens}{\percentchar\textvisiblespace}
String that prefixes the listing of \texttt{scriptFile}.

\option{environment}{enrionmentName}{none}
Name of \LaTeX{} environment (e.g., \texttt{quote})
  in which the listing is wrapped.

\subsubsection{Miscellaneous}
\option{verbose}{\texttt{true}/\texttt{false}}{\texttt{false}}
If \texttt{true}, the package logs every step it takes.

\subsection{Listings Styles}
Package
  \href
    {ftp://ftp.tex.ac.uk/tex-archive/macros/latex/contrib/listings/listings.pdf}
    {\textsf{listing}}
  is used for all listing both the executed shell
  commands and their output.
\subsubsection{Listings Style for Script File}

Style \verb+bashfulScript+ is used for displaying the executed shell
  commands (when option \texttt{script} is used).
The current definition of this style is:
\begin{verbatim}
  \lstdefinestyle{bashfulScript}{
    basicstyle=\ttfamily,
    keywords={},
    showstringspaces=false}


\end{verbatim}
Redefine this style to match your needs.

\subsubsection{Listings Style for Standard Output}
```

```
Style \verb+bashfulStdout+ is used for displaying the output of the
  executed shell
  commands (when option \texttt{stdout} is used).
The current definition is:
\begin{verbatim}
  % listings style for the stdoutFile, can be redefined by client
  \lstdefinestyle{bashfulStdout}{
    basicstyle=\sl\ttfamily,
    keywords={},
    showstringspaces=false
  }%

\end{verbatim}
Redefine this style to match your needs.


Style \verb+bashfulStderr+ is used for displaying the output of the
  executed shell
  commands (when option \texttt{stderr} is used).
\begin{verbatim}
  \lstdefinestyle{bashfulStderr}{
    basicstyle=\sl\ttfamily\color{red},
    keywords={},
    showstringspaces=false
  }
\end{verbatim}
Redefine this style to match your needs.

\section{Interaction with Other Packages}
This packages tries to work around a bug in \texttt{polyglossia}
  by which \verb+\texttt+ is garbled upon
  switching to languages which do not use the Latin alphabet.
Also, in case bidirectional \TeX{}ing is in effect,
  \bashful forces the listing to be left-to-right.


\section{History}
\begin{description}
\item[Version 0.91] Initial release.
\item[Version 0.92]
\begin{itemize}
  \item Added \texttt{ignoreExitCode},
    \texttt{ignoreStderr}, \texttt{stderr},
    \texttt{exitCodeFile} command options.

 \item
    Renamed \texttt{list} to \texttt{script}.
 \item
    Added \texttt{hide} and \texttt{dir} package options.
 \end{itemize}
\end{description}

\section{Future}
The following may get implemented some day.

\begin{enumerate}
\item \emph{Package options.} Currently all options are
    passed to the command itself.
\item \emph{A \texttt{clean} option.} This option
  will automatically erase files
  generated for storing the script, and its standard
  output and standard error streams.

\item \emph{A \texttt{noclobber} option.} This option
  will make this package safer, by reducing the risk
  of accidentally erasing existing files.
```

```
\end{enumerate}

\section{Acknowledgments}
The manner by which \verb+\bash+
  collects its arguments is based on that of
 \href
  {http://www.tn-home.de/Tobias/Soft/TeX/tobiShell.pdf}
  {\textsf{tobiShell}}.
Martin Scharrer tips on \TeX{} internals
  were invaluable.

\appendix
\section{Source of \texttt{\jobname.sty}}
  \lstinputlisting
    [    style=input,
         basicstyle=\scriptsize\ttfamily,
         numbers=left,
         stepnumber=10,
         firstnumber=1,
         numberfirstline=true,
         numberstyle=\scriptsize\rmfamily\bfseries
    ]
    {\jobname.sty}

\section{Source of \texttt{\jobname.tex}}
  \lstinputlisting
    [    style=input,
         basicstyle=\scriptsize\ttfamily,
         numbers=left,
         stepnumber=10,
         firstnumber=1,
         numberfirstline=true,
         numberstyle=\scriptsize\rmfamily\bfseries
    ]
    {\jobname.tex}
\end{document}
```