The bashful Package*

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

2012/03/12‡

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

1 Introduction

At the time I run this document through LaTeX, the temperature in Jerusalem, Israel, was *32℃*, while the weather condition was *clear*.

†mailto:yogi@cs.Technion.ac.IL

‡This document describes bashful V 0.94.

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.

Before I tell you how I generated this information, let me demonstrate the use of the bashful package for the purpose of incorporating the list of files in a folder into your output.

This simple LaTeX file generates a listing of all files in the `/usr` directory, using the UNIX `ls` command:

```
\documentclass{article}
\usepackage[a6paper]{geometry}
\usepackage{bashful}
\pagestyle{empty}
\begin{document}
The directories in my \texttt{/usr} directory are:
\bash[stdout]
ls -F /usr
\END
That's it!
\end{document}
```

The printed output of this file is then

The directories in my `/usr` directory are:

```
bin/
games/
include/
lib/
lib32/
local/
NX/
sbin/
share/
src/
```

That's it!

To generate the weather information, I wrote 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 Application for Teaching Programming

bashful primary application is for writing documents which describe computer programming. You can include the programs in your text, and have them compiled and executed as part of the LaTeX processing. To demonstrated I will first tell a simple story of writing, compiling and executing and a short program. Then, I will explain how I used the `\bash` command to not only tell the 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 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{article}
\usepackage[a6paper]{geometry}
\usepackage{bashful}
\pagestyle{empty}
\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{article}
\usepackage[a6paper]{geometry}
\usepackage{bashful}
\pagestyle{empty}
\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 send a "`q`" character to it to abort execution when the anticipated error occurs.

```
% yes q | xelatex -shell-esc minimal.tex | sed /texmf-dist/d
This is XeTeX, Version 3.1415926-2.4-0.9998 (TeX Live 2012/Debian)
 \write18 enabled.
entering extended mode
(./minimal.tex
LaTeX2e <2011/06/27>
Babel <v3.8m> and hyphenation patterns for english, dumylang, nohyphenation, lo
aded.
Document Class: article 2007/10/19 v1.4h Standard LaTeX document class
(/usr/share/texmf/tex/latex/xcolor/xcolor.sty
No file minimal.aux.
*geometry* driver: auto-detecting
*geometry* detected driver: xetex

Standard error not empty, and I was not instructed to ignore it.
>>>>>Your script file (minimal.sh) as I saw it was:
============================================================
echo "main(){return int;}" > error.c
cc error.c
============================================================
While the standard error stream file (minimal.stderr ) begins with
>>>>
>>>>error.c: In function main:
>>>>
but, you really ought to examine both files yourself!
! Your shell script failed....
\checkScriptErrors@BL ...r shell script failed...}
                                                  \BL@verbosetrue \logBL {Sw...
l.11 \END

? OK, entering \batchmode
```

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 main:
error.c:1:15: error: expected expression before int
```

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 Other Commands

`\bashStdout` After each execution of `\bash`, the macro `\bashStdout` is defined to entire contents of the standard output of the executed script.

For example, I can write

```
To obtain the following sentence:
\bash
uname -o
\END
\begin{quote}
``This document was prepared on \emph{\bashStdout}''
\end{quote}
```

To obtain the following sentence:

> "This document was prepared on *GNU/Linux*"

`\bashStderr` Similar to `\bashStderr`, except that it is defined to the standard error of the executed script. (Be ware that you must apply error tolerance flags to use this command, since normally, if the script generates anything to the standard error stream, LaTeX processing will halt, asking for your attention.)

`\bashScript` Similar to `\bashStdout` and `\bashStderr`, except that it is defined to the content of the most recent script.

`\splice` Shell commands passed to the `\splice` macro are executed in a similar fashion to commands enclosed between `\bash` and `\END`, but, in addition to this execution, bashful incorporates the standard output into the main file. For example, I can write

```
Here is a nice quote for you to remember.
\begin{quote}
\emph{\splice{fortune}}
\end{quote}
```

To obtain

> Here is a nice quote for you to remember.