The PracTEX Journal, 2012, No. 1
Article revision 2012/10/18

Bashful Writing and Active Documents

Joseph (Yossi) Gil*

Abstract In many ways, computerized typesetting still relies on metaphors drawn from the letterpress printing domain and is concerned largely with the production of documents printed on paper. Active documents is an emerging technology by which the product of computerized typesetting is more than an aesthetically pleasing composition of letters, words and punctuation characters broken into lines and pages. An active document offers modes of interaction with its reader, while the document itself may change its content in response to events taking place in the external world.

Bashful documents, the concept proposed by the LATEX bashful package (implemented as a wrapper around the `\write18` internal macro[a] extend this interaction to the *time of the document creation*. The author of a textbook on computer programming, may use bashful to automatically include in the text a transcript of a demonstration program, as it was executed in the time the document was authored. When writing a report on an experiment, a scientist may employ bashful to automatically execute the experiment, whenever the report text is run through LATEX, and even include the results in the output document. In fact, using bashful a document may include anything that can be computed, at the time of creation, by BASH, and the numerous Unix commands[b] it may invoke.

[a]In this document, I refer to TEX commands or macros, also called control sequences, solely as "macros".

[b]The term "commands" shall refer both to Unix programs which can be invoked from the command line prompt, and to BASH internal commands.

1 Introduction

At the time I run this document through LATEX, the temperature in Jerusalem, Israel, was *18°C*, while the weather condition was *clear*.

You may not care so much about these bits of truly ephemeral value, but you may be surprised that this information was produced automatically by the very

*yogi@CS.Technion.AC.IL

process of LaTeXing. The LaTeX source of this document included two sequences of commands, the first responsible for producing the temperature and the second for producing the weather condition. Each of these sequences was executed as the source was run through LaTeX; the output of this execution then replaced the sequence and then laid out as part of the text.

1.1 Dynamic Web Pages

It should be mentioned that the entire bashful process is similar to the method of generating dynamic web pages by "server-side scripting", including processors such as PHP, ASP, and Java server pages.

An author of a web site which employs PHP technology may start the creation of a page in his site by writing a simple text file named `good.php`, with the following content

```
<html>
  <body bgcolor="black" text="yellow">
    <?php
     $hour = date("G");
      if ($hour < 12)
        echo "Good morning, dear surfer!";
      else
        echo "Good evening, dear surfer!";
    ?>
  </body>
</html>
```

Just before this web page is delivered to the surfing user, the web server runs the page through a *PHP processor*, which executes all text enclosed between "<?php" and "?>" as a PHP program, replacing this text with the output of this program. The PHP program in this case is

```
$hour = date("G");
 if ($hour < 12)
   echo "Good morning, dear surfer!";
 else
   echo "Good evening, dear surfer!";
```

while the output of this program is either

```
Good morning, dear surfer!
```

or

```
Good evening, dear surfer!
```

Thus, depending on the time of day in which the request was made to the web server, file `good.php` will be sent to the user's browser as either

```
<html>
  <body bgcolor="black" text="yellow">
    Good morning, dear surfer!  </body>
</html>
```

or

```
<html>
  <body bgcolor="black" text="yellow">
    Good evening, dear surfer!  </body>
</html>
```

And, the display on the user's web browser will be as in Figure 1.

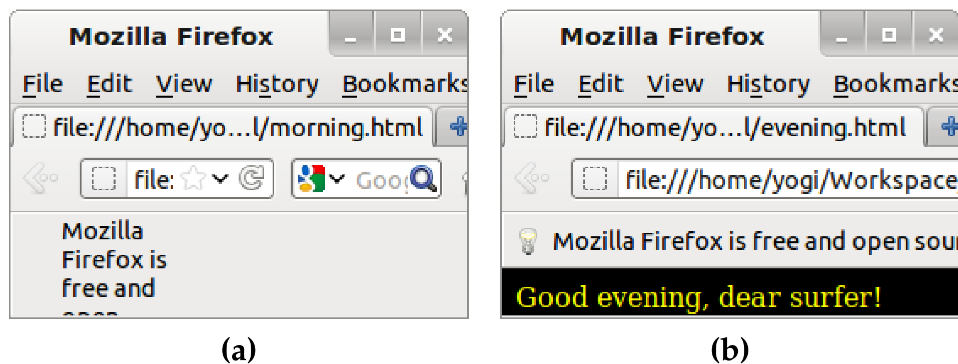


Figure 1: Two views of the same dynamic web page

1.2 Dynamic vs. Active vs. Bashful Documents

As we have seen a *dynamic document* is a document whose content may change just before it is delivered to the end user. *Active documents* go a step further, allowing the user to interact with document, by e.g., filling in forms included in the document, to click on buttons, navigate within and outside the document etc. This is made possible by technologies such as "client-side scripting", HTML forms and PDF interactive elements.

In contrast, *bashful documents* are characterized by the fact that their *generation* may yield different results, based on the time and the environment of the creation. For example, the weather report at the beginning of this document was produced by employing the bashful package to automatically make an HTTP connection to Google's weather service and then incorporate the result into the document.

We can also distinguish a class of *introspective documents*, whose content depends on meta-information of the contents. The sentence

> "*The document you are reading now was prepared from a single input file named* `00.tex`, *containing* 737 *lines and* 3790 *words of text.*"

is an example of an introspective content in this article.

The main application of the bashful package is in the preparation of computer programming articles and textbooks. Ideally, such a textbook would not use a single programming example without testing it. My inspiration in writing the bashful package dates to back to first edition of the seminal "The C Programming Language" book by Kernighan and Ritchie, widely known as K&R. The preface of this first edition tells its reader:

> *All examples have been tested directly from the text, which is in machine-readable form.*

And the second edition of K&R reiterates:

> *As before, all examples have been tested directly from the text, which is in machine-readable form.*

Bashful documents extend this idea a step further by executing and testing the programs directly by the processing of the text by LaTeX.

The article you are reading now is in itself a bashful document. The little PHP program you have just seen was generated and executed directly by the text processor, which was even employed to generate the screen captures in Figure 1.

This article also makes an example of an introspective document: It not only uses the bashful package a number of times to show programming examples; it also shows the reader what exactly I wrote in the input to produce this examples. And, as you may expect, the macros that I used are not shown to you by me manually copying the LaTeX input and then pasting it into a `verbatim` environment. Instead, the text processor is employed to introspectively fetch these macros from the input text. Clearly, one of the main applications of introspection is for writing documents that teach their readers how to use LaTeX.

Outline The remainder of this article is organized as follows. Section 2 explains the bashful basics and demonstrates how it can be used for writing computer programming textbooks. If you are interested in using bashful for writing documents discussing computer programming, this section, together with the bashful package documentation should suffice.

The process by which the weather report at the time of authoring was included in the beginning of this article is revealed in Section 3. Section 4 sheds some light on bashful internals, providing hints on dealing with errors.

As you read this article, note that document introspection is used extensively to show the actual input text in which the bashful package was used. I explain how this was done in Section 5.

For the sake of completeness, the full LaTeX source of this article is offered in Appendix A. Interested readers may examine this source to learn more, e.g., how Figure 1 was generated.

2 Bashful in Action

To demonstrate the bashful process, I now present a simple story of writing, compiling and executing and a simple program: Hello, World! in the C programming language: Then, I shall explain how the bashful package was employed to play the story live, that is, authoring the program, compiling it and executing it, all from within LaTeX.

2.1 "Hello, World!", Said Again

My story begins with the creation of a text file named `hello.c`, in which the program is stored.

```
% 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
```

(In the above, I used the `cat` Unix command to create a file in a manner known as *here document*, where my delimiting identifier was the string `EOF`.)

Once I have written my program, it is only natural to invoke the C compiler to translate it into an executable.

```
% cc hello.c
```

My little story reaches its climax when the program I created and compiled is executed, making sure that it prints the desired "Hello, World!" greeting.

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

2.2 Retrospection

The document you are reading was generated from a LaTeX input file whose name is `00.tex`. Examining file `00.tex`, you can see what I wrote in it to tell my story of the creation of file `hello.c` at the beginning of Section 2.1 above.

```
325  \subsection{``Hello, World!'', Said Again}\label{Section:story}
326  My story begins with the creation of a text file named
327    \texttt{hello.c}, in which the program is stored.
328  \bash[environment=quote,script]
329  cat << EOF > hello.c
330  /*
331  ** hello.c: My first C program; it prints
332  ** "Hello, World!", and dies.
333  */
334  #include <stdio.h>
335  int main()
336  {
337    printf("Hello, World!\n");
338    return 0;
339  }
340  EOF
341  \END
```

In doing so, all the text between the `\bash` (line 328) and `\END` (line 341) was copied by LaTeX to a temporary script file; this script is then sent for execution by Bash. The `script` option instructed the `\bash` macro to list this file in

the main document, while the `environment=quote` option instructed the `\bash` macro enclose the listing in a `quote` environment, i.e., between `\begin{quote}` and `\end{quote}`.

Note that two characters, "%␣", were automatically prepended to the script by the `\bash` macro. This is not an incident: %␣ is the default Bash prompt. Prepending it makes it clear to the reader that the script file is input to bash. (The `prefix` option to the `\bash` macro can be used to change this prefix string.)

To compile file `hello.c` that I just created, my `00.tex` included another `\bash` … `\END` pair.

```
347  Once I have written my program, it is only natural to invoke
348    the~C compiler to translate it into an executable.
349  \bash[environment=quote,script,stderr]
350  cc hello.c
351  \END
```

As before, in writing these I achieved two objectives: first, when LaTeX processed `00.tex`, it also invoked the C compiler to compile file `hello.c`, the file which I just created. Second, thanks to the `script` option, the command for compiling this program was included in the typeset version of this document. The `stderr` flag instructed the `\bash` macro to record the standard error stream of the script's execution, and layout this record further to the script. As can be seen above, the program I wrote was correct, the compilation process did not generate any error messages, and the standard error stream was left empty.

Finally, I executed the program I wrote. Here is another excerpt of `00.tex` showing how this was done.

```
352  My little story reaches its climax when the program I created
353    and compiled is executed, making sure that it prints the
354    desired ``Hello, World!'' greeting.
355  \bash[environment=quote,script,stdout]
356  ./a.out
357  \END
```

The `stdout` flag passed to the `\bash` macro above, instructs it to append to the script's listing the standard output stream that this execution produces, i.e., the string `Hello, World!`, as printed by program `a.out` to its standard output stream.

2.3 Input Processing

The \bash command is defined in package bashful. To make use of this package, I wrote in the preamble of 00.tex:

```
4   \usepackage[verbose,unique]{bashful}
```

The verbose boolean package option instructed the bashful package to be chatty, typing out for me a lot of information on what it does as the document is processed by LaTeX. The unique option instructs the package to use unique names, generated from the TeX's job name (\jobname) and the current line number. This option is essential for documents, such as the present document, in which the \bash command is used many times.

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-escape command line flag does that. To process my document, I typed, at the command line,

```
% xelatex -shell-escape 00.tex
```

3 Producing The Weather Information

A similar application of \bash to escape to shell was also used to produce the above Jerusalem weather report. However, since I wanted this information inlined in the text, I could not rely on the stdout flag to list the standard output of commands.

Instead, I wrote a series of shell commands that retrieve the current temperature, and another such series to obtain the current weather conditions. The command series to obtain the current temperature, was placed in a file named temperature.sh:

```
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 placed in a file named `condition.sh`

```
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'
```

I then executed the scripts `temperature.sh`, and `temperature.sh`, redirecting their output to files `temperature.tex` and `condition.tex`. All that remained was `\input` these two files in my `00.tex`.

```
90  At the time I run this document through
91    \href{http://www.latex-project.org/}{\LaTeX},
92    the \hypertarget{report}{temperature} in
93    \href{http://en.wikipedia.org/wiki/Jerusalem}{Jerusalem},
94    Israel, was~\emph{\input{temperature}\unskip\celsius},
95    while the weather condition was \emph{\input{condition}}\unskip.
```

I could have created files `temperature.sh` and `condition.sh` manually, but it made much more sense to both create and execute these using the `\bash` macro. For `temperature.sh`, I wrote in `00.tex`

```
67  \bash[scriptFile=temperature.sh,prefix={},stdoutFile=temperature.tex]
68  location=Jerusalem,Israel
69  server="http://www.Google.com/ig/api"
70  request="$server?weather=$location"
71  wget -q -O - $request |\
72  tr "<>" "\012\012" |\
73  grep temp_c |\
74  sed 's/[^0-9]//g'
75  \END
```

Passing the option `scriptFile=temperature.sh` instructed `\bash` to use the name `temperature.sh` to the script file it generated. The `prefix={}` option eliminated the Bash prompt that is normally prepended to the script. The third option, `stdoutFile=temperature.tex` saved the redirected output in a file named `temperature.tex`. Since none of the `script`, `stdout` and `stderr` flags was used, the execution of the script did not generate any text for typesetting by LaTeX.

What I wrote for generating `condition.sh`, executing it, and saving the output in

condition.tex was very similar.

```
78  \bash[scriptFile=condition.sh,prefix={},stdoutFile=condition.tex]
79  location=Jerusalem,Israel
80  server="http://www.Google.com/ig/api"
81  request="$server?weather=$location"
82  wget -q -O - $request |\
83  tr "<>" "\012\012" |\
84  grep "condition data" |\
85  head -n 1 |\
86  sed -e 's/^.*="//' -e 's/"\/*//' |\
87  tr 'A-Z' 'a-z'
88  \END
```

4 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 told the reader I did. More accurately, the `\bash` command, acting as my proxy, did it for me.

Luckily, the program I wrote was correct. But, if it was not, the `\bash` macro would have detected the error, and would have stopped the LaTeX process, indicating that the compilation did not succeed. To manage errors you should understand that the execution of the `\bash` macro involves the following steps:

1. collecting all text up to `\END`;
2. placing this text in a script file;
3. executing this script file, redirecting its standard output and its standard error streams to distinct files;
4. checking 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;
5. checking whether the file containing the standard error is empty, and if not, pausing execution after displaying an error message; and,
6. checking whether the file containing the exit code is empty, and if not, pausing execution after displaying an error message;

After the completion of these steps, the \bash macro may incorporate for typesetting three files in order: the script file (if the script flag is present), the standard output file (if the stdout flag is present), and then the standard error file (if the stderr flag is present).

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 error.tex which tries to use \bash to compile an incorrect C program. Since error.tex contains \END, I will have to author this file in three steps:

1. Creating the header of error.tex:
```
% cat << EOF > error.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 error.tex
```
% echo "\\END" >> error.tex
```
3. Finalizing error.tex
```
% cat << EOF >> error.tex
(I do not really expect the one-line
program generated above to compile.)
\end{document}
EOF
```

Let me verify that error.tex is what I expect it to be:
```
% cat error.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
(I do not really expect the one-line
program generated above to compile.)
\end{document}
```

I am now ready to run `error.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 error.tex | sed /texmf-dist/d
This is XeTeX, Version 3.1415926-2.3-0.9997.5 (TeX Live 2011)
 \write18 enabled.
entering extended mode
(./error.tex
LaTeX2e <2011/06/27>
Babel <v3.8m> and hyphenation patterns for english, dumylang, nohyphenation, ge
rman-x-2011-07-01, ngerman-x-2011-07-01, 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, serbianc, slovak, sloven
ian, spanish, swedish, turkish, turkmen, ukrainian, uppersorbian, welsh, loaded
.
Document Class: article 2007/10/19 v1.4h Standard LaTeX document class
*geometry* driver: auto-detecting
*geometry* detected driver: xetex

Standard error not empty. Here is how
file error.stderr begins:
>>>>error.c: In function main:
>>>>
but, you really ought to examine this file yourself!
! Your shell script failed....
\checkScriptErrors@BL ...r shell script failed...}
                                                  \BL@verbosetrue \logBL {Sw...
l.11 \END

? OK, entering \batchmode
```

(Observe that in the above I used the `sed` command to remove the mundane and lengthy logging messages of my `textmf` distribution.[1])

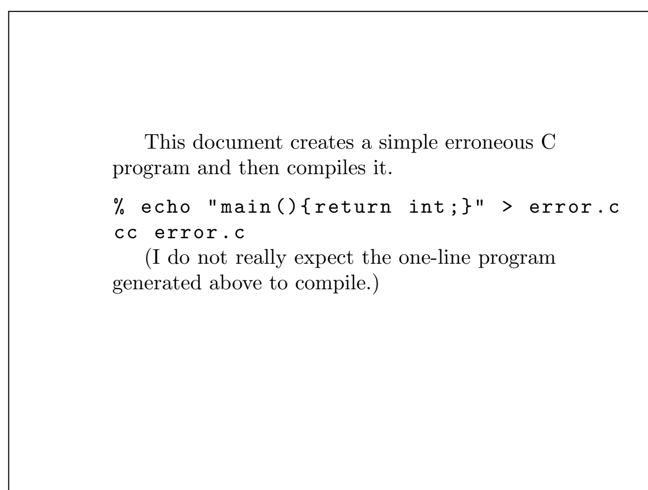
You can see that when LaTeX tried to process `error.tex`, it stopped execution

1. I also switched to a smaller font size, to allow the output to fit within the boundaries of the printed page.

while indicating that file `error.stderr` was not empty after the compilation. The first line of `error.stderr` was displayed, and I was advised to examine this file myself. Inspecting `error.stderr`, we see the C compiler error messages:

```
% cat error.stderr
error.c: In function main:
error.c:1:15: error: expected expression before int
```

The compilation error did not prevent LaTeX from typesetting my document. This final layout is presented in Figure 2. Note that the failure to compile `hello.c`, did not stop `\bash` from including this file in the source.



This document creates a simple erroneous C program and then compiles it.

```
% echo "main(){return int;}" > error.c
cc error.c
```

(I do not really expect the one-line program generated above to compile.)

Figure 2: File `error.pdf`

There are cases in which the author intends the executed script to generate errors. The `stderr` option to the `\bash` macro instructs it to *ignore* the exit code of the executed programww, and the fact that that output was generated to the standard error stream. Instead, `\bash` will include in its listing the contents of the standard error stream.

For example, to give you a taste of dealing with Bash script errors, I shall write below a passage expressing the frustration over Bash insisting on syntax trivialities.

```
638  A space must follow the opening square bracket; if not
639    \textsc{Bash} would not find the~``\verb+[+'' command.
640  The following script may seem correct on first sight, yet, the
641    error message it produces may seem weird to beginners.
```

```
642  \bash[prefix={},script,stdout,stderr]
643  if [2+2==5] ; then
644    echo "Freedom is the freedom to say that two plus two"
645    echo "make four. If that is granted, all else follows."
646  fi
647  \END
```

Indeed, newcomers to BASH may find conditionals confoudning. Annoying as it may sound, you have to remember rules such as: A space must follow the opening square bracket; if not BASH would not find the "`[`" command. The following script may seem correct on first sight, yet, the error message it produces may seem weird to beginners.

```
if [2+2==5] ; then
  echo "Freedom is the freedom to say that two plus two"
  echo "make four. If that is granted, all else follows."
fi
00@647.sh: line 1: [2+2==5]: command not found
```

The error message in the above was anticipated; it was included in the listing thanks to the `stderr` option. As explained, listing `stdout` instructs `\bash` to ignore the script's error code. LaTeX processing of `00.tex` does not stop as a result of this error.

5 Introspection

This article uses document introspection to show the actual input used to produce the examples. To achieve this, I used Unix commands to retrieve portions of `00.tex`, my input file, and `\input` these. As we shall see, the `sed` command proved instrumental in doing this.

Recall that at the beginning of Section 2.1, I wrote

> *My story begins with the creation of a text file named* `hello.c`*, in which the program is stored.*

Recall also that later, at the beginning of Section 2.2, I wrote

> *Examining file* `00.tex`*, you can see what I wrote in it to tell my story of the creation of file* `hello.c` *at the beginning of Section 2.1 above.*

And, immediately afterwards, I gave an excerpt of file `00.tex`.

To produce this excerpt, I applied the sed command to search in 00.tex. Specifically, what I wrote in 00.tex was the following

```
362  Examining file \me, you can see what I wrote in it to
363    tell my story of the creation of file \texttt{hello.c}
364    at the beginning of \autoref{Section:story} above.
365  \bash[stdout]
366  cat -n 00.tex | sed -n '/Said Again/,// { p
367    /END/q }'
368  \END
```

I used the cat command to number my input lines, and then the sed command to printing these lines, starting at the first line that contains the string "Said Again", and ending with line that contains the string "END".

My use of sed implies that file 00.tex includes the string "Said Again" at least twice. The first such occurrence was in the title of Section 2.1; the second occurrence was in the application of sed to introspectively search for the use of the \bash that followed this title. Subsequently, this document included several other occurrences of "Said Again" (including this sentence itself); but let us concentrate on the first two.

The search succeeded in finding the correct occurrence, since the search instructions occurred *after* it. You would need to apply a more sophisticated search in the case that you wish to present an input excerpt prior to its actual occurrence in the text. This was, for example, the case in the "taste" of Bash script errors offered in the previous section. I applied Gawk for this search. In case you are interested, the actual Unix pipeline I wrote was:

```
633  cat -n 00.tex|gawk '/A space must/{c++}c>1{print}/END/{if(c>1)exit}'
```

Acknowledgments The manner by which \bash collects its arguments is based on that of tobiShell. Martin Scharrer tips on TeX internals were invaluable in writing bashful.

A Source of 00.tex

```
1  \documentclass{pracjourn}\TPJrevision{2012}{10}{18}
    \TPJissue{2012}{1} \TPJcopyright{ }

   \usepackage[verbose,unique]{bashful}
   \usepackage{gensymb,graphicx,xspace,amsmath}
```

```
\newcommand\bashful{\textsf{bashful}\xspace}
\newcommand\Bash{\texttt{\textup{\textbackslash bash}}\xspace}
\newcommand\me{\texttt{\textup{\jobname.tex}}\xspace}
10
\lstdefinestyle{input}{basicstyle=\ttfamily\footnotesize,
    keywords={},upquote=true,extendedchars=false,
    showstringspaces=false,aboveskip=0pt,belowskip=0pt}
\lstdefinestyle{scriptsize}{style=input,basicstyle=\ttfamily\scriptsize}

% listings style for the script, standard output file, and standard error file.
\lstdefinestyle{bashfulScript}{style=input}
\lstdefinestyle{bashfulStdout}{style=input}
\lstdefinestyle{bashfulStderr}{style=input,
20        basicstyle=\ttfamily\footnotesize\color{red}}

\newcommand\listFile[1]{%
  \vspace{0.8em plus 0.3em minus 0.3em}%
  \lstinputlisting[style=input,frameround=fttt,frame=trBL]{#1}%
  \vspace{0.8em plus 0.3em minus 0.3em}}

\title{Bashful Writing and Active Documents}
\author{Joseph (Yossi) Gil\thanks{yogi@CS.Technion.AC.IL}}
\abstract{%
30 In many ways, computerized typesetting still relies on metaphors drawn from the
  \href{http://en.wikipedia.org/wiki/Letterpress_printing} {letterpress
  printing} domain and is concerned largely with the production of documents
  printed on paper.
Active documents is an emerging technology by which the product of computerized
  typesetting is more than an aesthetically pleasing composition of letters,
  words and punctuation characters broken into lines and pages.
An active document offers modes of interaction with its reader, while the
  document itself may change its content in response to events taking place in
  the external world.
40 \par
\emph{Bashful documents}, the concept proposed by the \LaTeX{}
  \href{http://ctan.org/tex-archive/macros/latex/contrib/bashful}{\bashful}
  package (implemented as a wrapper around the \texttt{\textbackslash write18}
  internal macro\footnote{%
    In this document, I refer to \TeX{} commands or macros, also called control
    sequences, solely as ``macros''.}\mbox{ }
  extend this interaction to the \emph{time of the document creation}.
The author of a textbook on computer programming, may use \bashful to
  automatically include in the text a transcript of a demonstration program, as
50 it was executed in the time the document was authored.
When writing a report on an experiment, a scientist may employ \bashful to
  automatically execute the experiment, whenever the report text is run through
  \LaTeX{}, and even include the results in the output document.
In fact, using \bashful a document may include anything that can be computed,
  at the time of creation, by
 \href{http://en.wikipedia.org/wiki/Bash\_(Unix\_shell)}{\textsc{bash}},
  and the numerous Unix commands\footnote{The term ``commands'' shall
    refer both to \href{http://en.wikipedia.org/wiki/Unix}{Unix} programs which
    can be invoked from the command line prompt, and to \textsc{Bash} internal
```

```
60      commands.}\mbox{ } it may invoke.
    }

    \begin{document}

    \maketitle

    \section{Introduction}
    \bash[scriptFile=temperature.sh,prefix={},stdoutFile=temperature.tex]
    location=Jerusalem,Israel
70  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[scriptFile=condition.sh,prefix={},stdoutFile=condition.tex]
    location=Jerusalem,Israel
80  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

90  At the time I run this document through
      \href{http://www.latex-project.org/}{\LaTeX},
      the \hypertarget{report}{temperature} in
      \href{http://en.wikipedia.org/wiki/Jerusalem}{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 value,
      but you may be surprised that this information was produced automatically
      by the very process of \LaTeX{}ing.
100 The \LaTeX{} source of this document included two sequences of commands, the
      first responsible for producing the temperature and the second for producing
      the weather condition.
    Each of these sequences was executed as the source was run through \LaTeX{};
      the output of this execution then replaced the sequence and then laid out as
      part of the text.

    \subsection{Dynamic Web Pages}
    It should be mentioned that the entire bashful process is similar to the method
      of generating \href{http://en.wikipedia.org/wiki/Dynamic_web_page}{dynamic
110   web pages} by ``\href{http://en.wikipedia.org/wiki/Server-side_scripting}
      {server-side scripting}'', including processors such as
      \href{http://en.wikipedia.org/wiki/PHP}{PHP},
```

```
      \href{http://en.wikipedia.org/wiki/Active_Server_Pages}{ASP}, and
      \href{http://en.wikipedia.org/wiki/JavaServer_Pages}{Java server pages}.

    An author of a web site which employs PHP technology may start the creation of
     a page in his site by writing a simple text file named \texttt{good.php}, with
      the following content
    \bash[scriptFile=good.sh]
120 cat << EOF > good.php
    <html>
      <body bgcolor="black" text="yellow">
        <?php
         \$hour = date("G");
          if (\$hour < 12)
            echo "Good morning, dear surfer!";
          else
            echo "Good evening, dear surfer!";
        ?>
130   </body>
    </html>
    EOF
    \END
    \listFile{good.php}

    Just before this web page is delivered to the surfing user, the web server runs
      the page through a \emph{PHP processor}, which executes all text enclosed
      between~``\texttt{<?php}'' and ``\texttt{?>}'' as a PHP program, replacing
      this text with the output of this program.
140 The PHP program in this case is
    \bash[stdout,stdoutFile=good.html,scriptFile=good.php.sh]
    sed -n "/hour/,/evening/ p" good.php
    \END
    while the output of this program is either
    \bash[stdout,stdoutFile=morning.out,scriptFile=morning.sh]
    grep morning good.php | sed -e s/echo// -e "s/;//" -e "s/\"//g"
    \END
    or
    \bash[stdout,stdoutFile=evening.out,scriptFile=evening.sh]
150 grep evening good.php | sed -e s/echo// -e "s/;//" -e "s/\"//g"
    \END

    Thus, depending on the time of day in which the request was made to the web
      server, file \texttt{good.php} will be sent to the user's browser as either
    \bash[scriptFile=morning.html.sh]
    php good.php | sed s/evening/morning/ > morning.html
    \END
    \listFile{morning.html}
    or
160 \bash[scriptFile=evening.html.sh]
    php good.php | sed s/morning/evening/ > evening.html
    \END
    \listFile{evening.html}

    And, the display on the user's web browser will be
```

```
    as in \autoref{Figure:firefox}.

  \begin{figure}[!h]
  \bash[scriptFile=firefox.sh,ignoreStderr]
170 rm evening.png morning.png
  firefox=`pgrep firefox`
  if [ -n "$firefox" ]; then
    wmctrl -c firefox
    kill $firefox
    killall firefox
  fi
  firefox -CreateProfile delme
  firefox -P delme morning.html &
  sleep 2
180 wmctrl -r "Mozilla Firefox" -b remove,maximized_vert,maximized_horz
  wmctrl -r "Mozilla Firefox" -e 0,0,0,270,150
  sleep 1
  scrot -u morning.png
  wmctrl -c firefox
  killall firefox
  firefox -P delme evening.html &
  sleep 2
  wmctrl -r "Mozilla Firefox" -b remove,maximized_vert,maximized_horz
  wmctrl -r "Mozilla Firefox" -e 0,0,0,270,150
190 scrot -u evening.png
  wmctrl -c firefox
  killall firefox
  if [ -n "$firefox" ]; then
    echo $firefox
    firefox -P default &
  fi
  \END
  \centering
  \begin{tabular}{cc}
200 \includegraphics[width=0.4\textwidth]{morning.png}
  &
  \includegraphics[width=0.4\textwidth]{evening.png}
  \\
  \bfseries (a) & \bfseries (b)
  \end{tabular}
  \caption{Two views of the same dynamic web page}
  \label{Figure:firefox}
  \label{firefox}
  \end{figure}
210
  \subsection{Dynamic vs. Active vs. Bashful Documents}
  As we have seen a \emph{dynamic document} is a document whose content may
    change just before it is delivered to the end user.
  \emph{Active documents} go a step further, allowing the user to interact with
    document, by e.g., filling in forms included in the document, to click on
    buttons, navigate within and outside the document etc.
  This is made possible by technologies such as
    \href{http://en.wikipedia.org/wiki/Client\_side\_scripting}{``client-side
```

```
    scripting''}, \href{http://en.wikipedia.org/wiki/HTML\_forms}{HTML forms}
220 and \href{http://en.wikipedia.org/wiki/%
    Portable\_Document\_Format\#Interactive\_elements}
    {PDF interactive elements}.

  In contrast, \emph{bashful documents} are characterized by the fact that their
   \emph{generation} may yield different results, based on the time and the
    environment of the creation.
  For example, the weather report at the \hyperlink{report}{beginning} of this
    document was produced by employing the \bashful package to automatically make
    an \href{http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol}{HTTP}
230 connection to \href{http://www.Google.com/support/forum/p/ apps-apis/thread?%
    tid=0c95e45bd80def1a&hl=en}{Google's weather service} and then incorporate
    the result into the document.

  We can also distinguish a class of \emph{introspective documents}, whose
    content depends on meta-information of the contents.  The sentence
  \begin{quote}
  \bash
  wc -l 00.tex | sed s/00.tex// > lines.tex
  wc -w 00.tex | sed s/00.tex// > words.tex
240 \END
  ``\textsl{The document you are reading now was prepared from a single input file
    named \me, containing \emph{\input{lines}\unskip} lines and
    \emph{\input{words}\unskip} words of text.}''
  \end{quote}
  is an example of an introspective content in this article.

  The main application of the \bashful package is in the preparation of computer
    programming articles and textbooks.
  Ideally, such a textbook would not use a single programming example without
250 testing it.
  My inspiration in writing the \bashful package
    dates to back to first edition of the seminal
    \href{http://en.wikipedia.org/wiki/The_C_Programming_Language}
      {``The C Programming Language''} book by
   \href{http://en.wikipedia.org/wiki/Brian_Kernighan}
        {Kernighan} and
   \href{http://en.wikipedia.org/wiki/Dennis\_Ritchie}
      {Ritchie}, widely known as K\&R.
  The preface of this first edition tells its reader:
260 \begin{quote}
    \textit{All examples have been tested directly from the text,
    which is in machine-readable form.}
  \end{quote}
  And the second edition of K\&R reiterates:
  \begin{quote}
    \textit{As before,
        all examples have been tested directly from the text,
          which is in machine-readable form.}
  \end{quote}
270 Bashful documents extend this idea a step further by executing and testing the
    programs directly by the processing of the text by \LaTeX.
```

```
    The article you are reading now is in itself a bashful document.
    The little PHP program you have just seen was generated and executed directly
      by the text processor, which was even employed to generate the screen captures
      in \autoref{Figure:firefox}.

    This article also makes an example of an introspective document:
    It not only uses the \bashful package a number of times to show programming
280   examples; it also shows the reader what exactly I wrote in the input to
      produce this examples.
    And, as you may expect, the macros that I used are not shown to you by me
      manually copying the \LaTeX{} input and then pasting it into a
      \texttt{verbatim} environment.
    Instead, the text processor is employed to introspectively fetch these macros
      from the input text.
    Clearly, one of the main applications of introspection is for writing documents
      that teach their readers how to use \LaTeX{}.

290 \renewcommand\sectionautorefname{Section}
    \renewcommand\subsectionautorefname{Section}
    \paragraph{Outline}
    The remainder of this article is organized as follows.
    \autoref{Section:action} explains the \bashful basics and demonstrates how it
      can be used for writing computer programming textbooks.
    If you are interested in using \bashful for writing documents discussing
      computer programming, this section, together with the \bashful package
      documentation should suffice.

300 The process by which the weather report at the time of authoring was included
      in the \hyperlink{report}{beginning} of this article is revealed in
      \autoref{Section:weather}.
    \autoref{Section:errors} sheds some light on \bashful internals, providing hints
      on dealing with errors.

    As you read this article, note that document introspection is used extensively
      to show the actual input text in which the \bashful package was used. I
      explain how this was done in \autoref{Section:introspection}.

310 For the sake of completeness, the full \LaTeX{} source of this article is
      offered in \autoref{Section:source}.
    Interested readers may examine this source to learn more, e.g., how
      \autoref{Figure:firefox} was generated.

    \section{Bashful in Action}\label{Section:action}
    To demonstrate the bashful process, I now present a simple story of writing,
      compiling and executing and a simple program:
      \href{http://en.wikipedia.org/wiki/Hello_world_program}{Hello, World!} in the
      \href{http://en.wikipedia.org/wiki/C_(programming_language)}{C programming
320   language}:
    Then, I shall explain how the \bashful package was employed to play the story
      live, that is, authoring the program, compiling it and executing it, all from
      within \LaTeX{}.
```

```
    \subsection{``Hello, World!'', Said Again}\label{Section:story}
    My story begins with the creation of a text file named
      \texttt{hello.c}, in which the program is stored.
    \bash[environment=quote,script]
    cat << EOF > hello.c
330 /*
    ** hello.c: My first C program; it prints
    ** "Hello, World!", and dies.
    */
    #include <stdio.h>
    int main()
    {
      printf("Hello, World!\n");
      return 0;
    }
340 EOF
    \END
    (In the above, I used the \href{http://en.wikipedia.org/wiki/Cat\_(Unix)}
      {\texttt{cat}} Unix command to create a file in a manner known as
      \href{http://en.wikipedia.org/wiki/Here_document}{\emph{here document}}, where
       my delimiting identifier was the string \texttt{EOF}.)

    Once I have written my program, it is only natural to invoke
      the~C compiler to translate it into an executable.
    \bash[environment=quote,script,stderr]
350 cc hello.c
    \END
    My little story reaches its climax when the program I created
      and compiled is executed, making sure that it prints the
      desired ``Hello, World!'' greeting.
    \bash[environment=quote,script,stdout]
    ./a.out
    \END

    \subsection{Retrospection}\label{Section:retrospection}
360 The document you are reading was generated from a \LaTeX{} input file whose name
      is \me.
    Examining file \me, you can see what I wrote in it to
      tell my story of the creation of file \texttt{hello.c}
      at the beginning of \autoref{Section:story} above.
    \bash[stdout]
    cat -n 00.tex | sed -n '/Said Again/,// { p
      /END/q }'
    \END
    % Applies sed to introspectively search the input
370
    \bash
    cat -n 00.tex | sed -n '/Said Again/,// {
      /\\bash/ { =
        q
      }
    }'
    \END\let\firstBash\bashStdout
```

```
\bash
cat -n 00.tex | sed -n '/Said Again/,// {
  /END/ {
    =
    q
  }
}'
\END\let\lastBash\bashStdout

In doing so, all the text between the \Bash (line \firstBash) and \verb+\END+
  (line \bashStdout) was copied by \LaTeX{} to a temporary script file; this
  script is then sent for execution by \textsc{Bash}.
The \texttt{script} option instructed the \Bash macro to list this file in the
  main document, while the \texttt{environment=quote} option instructed the
 \Bash macro enclose the listing in a \texttt{quote} environment, i.e., between
 \verb+\begin{quote}+ and \verb+\end{quote}+.

Note that two characters, ``\verb*+% +'', were automatically prepended to the
  script by the \Bash macro.
This is not an incident: \verb*+% + is the default \textsc{Bash}
  \href{http://en.wikipedia.org/wiki/Command-line_interface\#Command_prompt}
  {prompt}.
Prepending it makes it clear to the reader that the script file is input to
  \textsc{bash}.
(The \texttt{prefix} option to the \Bash macro can be used to change this
  prefix string.)

To compile file \texttt{hello.c} that I just created, my \texttt{00.tex}
  included another \Bash \ldots \verb+\END+ pair.
\bash[stdout]
cat -n 00.tex | sed -n '/Once I have written/,// { p
  /END/q }'
\END
As before, in writing these I achieved two objectives: first, when \LaTeX{}
  processed \me, it also invoked the~C compiler to compile file
  \texttt{hello.c}, the file which I just created.
Second, thanks to the \texttt{script} option, the command for compiling this
  program was included in the typeset version of this document.
The \texttt{stderr} flag instructed the \Bash macro to record the standard
  error stream of the script's execution, and layout this record further to the
  script.
As can be seen above, the program I wrote was correct, the compilation process
  did not generate any error messages, and the standard error stream was left
  empty.

Finally, I executed the program I wrote.
Here is another excerpt of \me showing how this was done.
\bash[stdout]
cat -n 00.tex | sed -n '/climax/,// { p
  /END/q }'
\END
The \texttt{stdout} flag passed to the \Bash macro above, instructs it to
  append to the script's listing the standard output stream that this execution
```

```
      produces, i.e., the string \texttt{Hello, World!}, as printed by program
      \texttt{a.out} to its standard output stream.

    \subsection{Input Processing}
    The \Bash command is defined in package \bashful.
    To make use of this package, I wrote in the preamble of \me:
    \bash[stdout]
    cat -n 00.tex | sed -n '/bashful/,// { p
      /bashful/q }'
440 \END
    The \texttt{verbose} boolean package option instructed the \bashful package to
      be chatty, typing out for me a lot of information on what it does as the
      document is processed by \LaTeX{}.
    The \texttt{unique} option instructs the package to use unique names,
      generated from the \TeX{}'s job name (\verb+\jobname+) and the
      current line number.
    This option is essential for documents, such as the present document,
      in which the \verb+\bash+ command is used many times.

450 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-escape} command line flag does that.
    To process my document, I typed, at the command line,
    \begin{quote}
      \texttt{\% xelatex -shell-escape \me}
    \end{quote}
460
    \section{Producing The Weather Information} \label{Section:weather}
    A similar application of \Bash to escape to shell was also used to
      produce the above Jerusalem weather report.
    However, since I wanted this information inlined in the text, I could not rely
      on the \texttt{stdout} flag to list the standard output of commands.

    Instead, I wrote a series of shell commands that retrieve the current
      temperature, and another such series to obtain the current weather conditions.
    The command series to obtain the current temperature, was placed in a file
470   named \texttt{temperature.sh}:
    \listFile{temperature.sh}
    while the weather condition was placed in a file named \texttt{condition.sh}
    \listFile{condition.sh}

    I then executed the scripts \texttt{temperature.sh}, and
      \texttt{temperature.sh}, redirecting their output to files
      \texttt{temperature.tex} and \texttt{condition.tex}.
    All that remained was \verb+\input+ these two files in my \texttt{\jobname.tex}.
    \bash[stdout,stdoutFile=weather.tex]
480 cat -n 00.tex | sed -n '/At the time I run/,// { p
      /while the weather condition/q }'
    \END
```

```
    I could have created files \texttt{temperature.sh} and \texttt{condition.sh}
      manually, but it made much more sense to both create and execute these using
      the \Bash macro.
    For \texttt{temperature.sh}, I wrote in \texttt{\jobname.tex}
    \bash[stdout,stdoutFile=temperature.lst]
    cat -n 00.tex | sed -n '/temperature.sh/,// { p
490   /END/q }'
    \END
    \noindent
    Passing the option \texttt{scriptFile=temperature.sh} instructed \Bash to
      use the name \texttt{temperature.sh} to the script file it generated.
    The \verb+prefix={}+ option eliminated the \textsc{Bash} prompt that is normally
            prepended to the script.
    The third option, \verb+stdoutFile=temperature.tex+ saved the
      redirected output in a file named \texttt{temperature.tex}.
    Since none of the \texttt{script}, \texttt{stdout} and \texttt{stderr} flags
500   was used, the execution of the script did not generate any text for
      typesetting by \LaTeX{}.

    \noindent What I wrote for generating \texttt{condition.sh},
      executing it, and saving the output in \texttt{condition.tex}
      was very similar.
    \bash[stdout]
    cat -n 00.tex | sed -n '/condition.sh/,// { p
      /END/q }'
    \END
510
    \section{Dealing with Errors}\label{Section:errors}
    Using \bashful{} to demonstrate my \emph{Hello, World!} program, made sure that
      the story I told is accurate:
    I really did everything I told the reader I did.
    More accurately, the \Bash command, acting as my proxy, did it for me.

    Luckily, the program I wrote was  correct.
    But, if it was not, the \Bash macro would have detected the error, and
      would have stopped the \LaTeX{} process, indicating that the compilation did
520   not succeed.
    To manage errors you should understand that the execution of the \Bash
      macro involves the following steps:
    \begin{enumerate}
    \item collecting all text up to \verb+\END+;
    \item placing this text in a script file;
    \item executing this script file, redirecting its standard output
          and its standard error streams to distinct files;
    \item checking 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
530   distinct file;
    \item checking whether the file containing the standard error is empty, and if
      not, pausing execution after displaying an error message; and,
    \item checking whether the file containing the exit code is empty, and if not,
      pausing execution after displaying an error message;
    \end{enumerate}
    After the completion of these steps, the \Bash macro may incorporate for
```

```
  typesetting three files in order: the script file (if the \text{script} flag
  is present), the standard output file (if the \text{stdout} flag is present),
  and then the standard error file (if the \text{stderr} flag is present).
540
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{error.tex} which
  tries to use \Bash to compile an incorrect~C program.
Since \texttt{error.tex} contains \verb+\END+, I will have to author this file
  in three steps:
\begin{enumerate}
\item Creating the header of \texttt{error.tex}:
\bash[script]
550 cat << EOF > error.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
560 cc error.c
EOF
\END
\item Adding \verb+\END+ to \texttt{error.tex}
\bash[script]
echo "\\END" >> error.tex
\END
\item Finalizing \texttt{error.tex}
\bash[script]
cat << EOF >> error.tex
570 (I do not really expect the one-line
program generated above to compile.)
\end{document}
EOF
\END
\end{enumerate}
Let me verify that \texttt{error.tex} is what I expect it to be:
\bash[script,stdout]
cat error.tex
\END
580
I am now ready to run \texttt{error.tex} through \LaTeX{}, but since I will not
  run the \texttt{latex} command myself, I will send a ``\texttt{q}'' character
  to it to abort execution when the anticipated error occurs.

\lstdefinestyle{bashfulScript}{style=scriptsize}
\lstdefinestyle{bashfulStdout}{style=scriptsize}
\bash[script,stdout]
yes q | xelatex -shell-esc error.tex | sed /texmf-dist/d
\END
```

```
590 \lstdefinestyle{bashfulScript}{style=input}
    \lstdefinestyle{bashfulStdout}{style=input}

    (Observe that in the above I used the
      \href{http://www.gnu.org/software/sed/manual/sed.html}{\texttt{sed}}
      command to remove the mundane and lengthy logging messages of my
      \texttt{textmf} distribution.%
                  \footnote{I also switched to a smaller font size, to allow
                          the output to fit within the boundaries of the printed page.})

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

    The compilation error did not prevent \LaTeX{} from typesetting my document.
610 This final layout is presented in \autoref{Figure:error}.
    Note that the failure to compile \texttt{hello.c}, did not stop \Bash
      from including this file in the source.

    \begin{figure}[!h]
    \begin{center}
      \fbox{\includegraphics[scale=0.8,trim=0 200 0 0]{error.pdf}}
    \end{center}
    \caption{File \texttt{error.pdf}}\label{Figure:error}
    \end{figure}
620
    There are cases in which the author intends the executed script to generate
      errors.
    The \texttt{stderr} option to the \Bash macro instructs it to
      \emph{ignore} the exit code of the executed programww, and the fact that that
      output was generated to the standard error stream.
    Instead, \Bash will include in its listing the contents of the standard
      error stream.

    For example, to give you a taste of dealing with \textsc{Bash} script errors, I
630   shall write below a passage expressing the frustration over \textsc{Bash}
      insisting on syntax trivialities.
    \bash[stdout]
    cat -n 00.tex|gawk '/A space must/{c++}c>1{print}/END/{if(c>1)exit}'
    \END

    Indeed, newcomers to \textsc{Bash} may find conditionals confoudning.
    Annoying as it may sound, you have to remember rules such as:
    A space must follow the opening square bracket; if not
      \textsc{Bash} would not find the~``\verb+[+'' command.
640 The following script may seem correct on first sight, yet, the
      error message it produces may seem weird to beginners.
    \bash[prefix={},script,stdout,stderr]
```

```
if [2+2==5] ; then
  echo "Freedom is the freedom to say that two plus two"
  echo "make four. If that is granted, all else follows."
fi
\END

The error message in the above was anticipated; it was included
  in the listing thanks to the \texttt{stderr} option.
As explained, listing \texttt{stdout} instructs \Bash to ignore
  the script's error code.
\LaTeX{} processing of \texttt{\jobname.tex}
  does not stop as a result of this error.

\section{Introspection}
\label{Section:introspection}
This article uses document introspection to show the actual input used to
  produce the examples.
To achieve this, I used Unix commands to retrieve portions of
  \texttt{\jobname.tex}, my input file, and \verb+\input+ these.
As we shall see, the \texttt{sed} command proved instrumental in doing this.

Recall that at the beginning of \autoref{Section:story}, I wrote
\bash[stdoutFile=begins.tex]
cat 00.tex | sed -n '/begins/,// { p
  /stored/q }'
\END
\begin{quote}
  \textit{\input{begins.tex}}
\end{quote}
Recall also that later, at the beginning of
  \autoref{Section:retrospection}, I wrote
\bash[stdoutFile=examining.tex]
cat 00.tex | sed -n '/Examining/,// { p
  /above/q }'
\END
\begin{quote}
  \textit{\input{examining.tex}}
\end{quote}
And, immediately afterwards, I gave an excerpt of file \texttt{\jobname.tex}.

To produce this excerpt, I applied the \texttt{sed}
  command to search in \texttt{\jobname.tex}.
Specifically, what I wrote in \texttt{\jobname.tex} was the following
\bash[stdout]
cat -n 00.tex | sed -n '/Examining/,// {
  /introspectively search the input/q
  p }'
\END
I used the \texttt{cat} command to number my input lines, and then the
  \texttt{sed} command to printing these lines, starting at the first line that
  contains the string ``Said Again'', and ending with line that contains the
  string ``END''.
```

Line numbers in margin: 650, 660, 670, 680, 690

```
    My use of \texttt{sed} implies that file \texttt{\jobname.tex} includes
      the string ``Said Again'' at least twice.
    The first such occurrence was in the title of \autoref{Section:story}; the
      second occurrence was in the application of \texttt{sed} to introspectively
700   search for the use of the \Bash that followed this title.
    Subsequently, this document included several other occurrences of
     ``Said Again'' (including this sentence itself); but let us concentrate
       on the first two.

    The search succeeded in finding the correct occurrence, since the search
      instructions occurred \emph{after} it.
    You would need to apply a more sophisticated search in the case that you
      wish to present an input excerpt prior to its actual occurrence in the text.
    This was, for example, the case in the ``taste'' of \textsc{Bash} script errors
710   offered in the previous section.
    I applied \href{http://www.gnu.org/software/gawk/}{Gawk}
      for  this search.
    In case you are interested, the actual \href{http://en.wikipedia.org/wiki/%
      Pipeline\_(Unix)} {Unix pipeline} I wrote was:
    \bash[stdout]
    cat -n 00.tex | sed -n '/gawk/,// { p
      q }'
    \END

720 \paragraph{Acknowledgments}
    The manner by which \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 in writing \bashful.
    \appendix
    \section{Source of \texttt{\jobname.tex}}
    \label{Section:source}
      \lstinputlisting
        [    style=input,
             basicstyle=\scriptsize\ttfamily,
730          numbers=left,
             stepnumber=10,
             firstnumber=1,
             numberfirstline=true,
             numberstyle=\scriptsize\rmfamily\bfseries
        ]
        {\jobname.tex}
    \end{document}
```