LiPPGen: A presentation generator for literate-programming-based teaching

Hans-Georg Eßer

Abstract

Literate programming techniques can be used as a teaching method — not only in book form, but also for lectures in the classroom. I present a tool which helps instructors transform their literate programs into lecture presentations: LiPPGen, the Literate-Programming-based Presentation Generator, takes a standard literate program (with LaTeX as the documentation language) as input and lets the instructor comfortably generate presentation slides for each code chunk. It then assembles the provided slide texts and the code chunks and turns them into a browser-based presentation.

LiPPGen offers unique features in comparison to standard presentation programs (such as PowerPoint) in that code chunks may be larger than the space on a slide permits: if so, the code can be scrolled during the presentation. The code is also presented with syntax highlighting using simple regular-expression-based rules. Currently, C and Python are supported.

This article describes both the features and usage of LiPPGen and provides an example, showing a small literate program (the implementation of a component of an educational operating systems) and its transformation into lecture slides.

LiPPGen is available under an open source license so that others who use literate programming in an instructional environment can also use the software and modify it to their needs.

1 Introduction

Literate programming [5] is a programming technique invented by D. E. Knuth which lets developers create source code and documentation in one "literate program" from which both well-readable documentation and compilable source files can be extracted. It can be used to create textbooks on any computer science topic that involves presenting and explaining larger program code blocks. This approach has been used by a few authors, including Knuth himself, when he published the TeX source code [6], but also more recently, for example by Pharr and Humphreys who explain the art of 3D rendering in their book [10]. Additional literate-programming-based textbooks are mentioned in the "Books: Applies Literate Programming" section of the `literateprogramming.com` link list [7].

One of the advantages of literate programming over other development styles is that the order of presentation does not depend on syntactical constraints. For a developer this means that the original creative process can be recorded in the literate program, allowing both top-down and bottom-up approaches. Instructors can base the presentation on didactic considerations, for example, they can first give function definitions and structure declarations in an incomplete form (when the audience does not have the required knowledge to understand the full versions) and later extend them when the missing information has been taught.

Since a book is not helpful in a classroom setting, the question arises of how an instructor might create lecture slides from a literate program. While it is possible to copy and paste fragments from the literate program into a presentation and manually add text, such a procedure is tedious and will not always lead to good results. Also, whenever the author modifies the original literate program, he or she must also manually update the slides. Until now, there has been no software to aid the instructor in creating a literate-programming-based presentation.

2 LiPPGen features

LiPPGen [2] lets you select a part (or several parts) of a literate document by marking blocks with `begin` and `end` comments. When running the tool on such a file, it creates an HTML file and opens it in the browser (Figure 1). The page shows code chunks and documentation blocks separately (with most LaTeX code either stripped or converted to equivalent HTML), and for each code chunk you can enter some descriptive text in an HTML editor field sitting next to the code chunk. (The program displays the documentation parts as well so that you can easily decide what information to pick for the slide

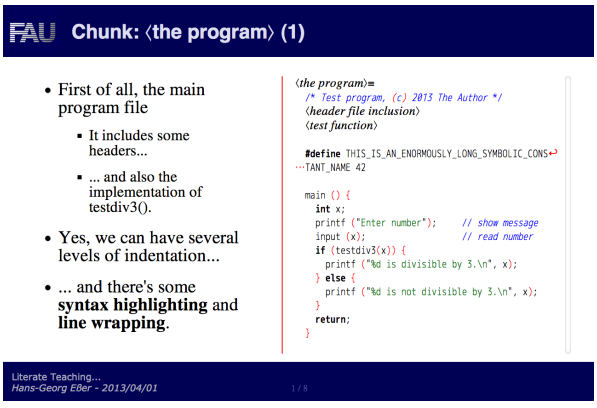


Figure 1: LiPPGen lets you convert literate programs into presentations, with large parts of the process being fully automatic.

content.) The editor allows for simple markup, such as enumerations, bullet lists, bold, and italics.

When you've entered all the information, you can send the data back to the program (it supplies a simple HTTP server for just this purpose), and then LiPPGen creates an HTML presentation file with the code chunks and your added input. Finally it opens the new presentation in the browser.

Additional features are:

- You can repeat the editing process several times; data entered in a preceding program run remains available.
- The presentation shows chunk names in a way that is similar to the formatting in a traditional literate program (e. g., ⟨*name*⟩), both for the definition (as in ⟨*name*⟩≡... and ⟨*name*⟩+≡...) as well as for occurrences in other code chunks.
- LiPPGen recognizes repeated (i. e., continuing) definitions of code chunks. The first one is always shown as ⟨*name*⟩≡, whereas the following ones use ⟨*name*⟩+≡. Also, the chunk names are used as slide titles, and if a chunk occurs on several slides, LiPPGen increments a counter.
- Simple syntax highlighting (via regular expressions) is available for C and Python source code, so there's a bit of pretty printing. You can easily extend this to other languages.
- As part of the syntax highlighting LiPPGen also breaks code lines which are too long. The continuation is shown via an arrow character at the end (similar to the display in the XEmacs editor) and there are dots at the beginning of the following line. That way it is clear where a line begins and ends, without a need to introduce line numbers.
- When you give the presentation, you can scroll code chunks up and down using the cursor keys (for code chunks which are longer than the slide permits). Each code chunk "remembers" the current scrolling position, so when you later return to a slide, the display of the code chunk is as it was when you last left the slide.

3 Implementation

When attempting to convert a document which is basically in LaTeX syntax — though in its extended Literate Programming form — the natural choice for slide creation would be to stick with LaTeX and use one of the available LaTeX document classes for presentations, such as `beamer` [13]. However, the end result of any approach using LaTeX will be a PDF document, and such documents are static.

In contrast, HTML allows elements on a page to be scrollable, and this feature comes in handy when we want to show code which exceeds the available space on a slide.

3.1 Recycling: Use what's there

Classically, open source developers are too lazy to reinvent the wheel, and so at the beginning I checked for available tools which might be able to reduce my own development efforts. I found two very helpful programs:

- The "Simple Standards-based Slide Show System" (S5) contains CSS files which let users create complete presentations in single HTML files [8]. Adding a slide is as simple as writing

  ```
  <div class="slide">
  <h1>Slide Title</h1>
  ...
  </div>
  ```

  in the source file, and bullet items need no more than standard HTML lists (`<ul><li>...</li> </ul>`).

  A LiPPGen presentation looks a lot like a standard S5 presentation, except for the added literate programming bits.
- The "NicEdit Inline Content Editor" [4] is a JavaScript program that provides an HTML editor which can be embedded in HTML pages. It is customizable, and for LiPPGen I've disabled most of the available buttons, since they are not needed.

3.2 The power of Python

Python comes with several useful libraries and allows the on-the-fly implementation of a simple web server. (This is true for many other script languages, but I know Python best — the simple reason for choosing it.) We need one for accepting the user's input on the web form, and Python's `socket` module let me integrate the web server into the program.

The complete `lippgen` script is only about 700 lines long, and these few lines of code handle parsing the literate program document, generating the HTML form, accepting the user input, and assembling the final HTML files with syntax highlighting, line breaking and other stuff.

3.3 Some JavaScript as well

In order to allow scrolling of the code chunks, I had to slightly modify the JavaScript code that is part of the S5 system. In brief, I've given HTML names to all code chunks and changed the key-press event code so that [Cursor up] and [Cursor down] scroll the currently displayed code chunk up and down.

I also modified S5's `default/pretty.css` file so that the standard font for listings (`tt`) is `M+ 1m` [9]

since this font runs narrower than the standard Courier type fonts.

4 LiPPGen tutorial

To try out LiPPGen yourself, download the software and install it; then pick a sample literate program and use `lippgen`. We'll describe the process here.

4.1 Installing LiPPGen

Do the following to install the program:

1. Unpack the archive and copy the `lippgen` and `lippgen-sanitize` files to some directory in your path (e. g., `/usr/bin/` or `~/bin`). Create `/usr/share/lippgen/` and recursively copy the `lippgen.d/` directory into that new directory. If you cannot write in `/usr/share` you can pick some other location but will then have to modify the assignment

   ```
   LIPPGEN_D = \
     "/usr/share/lippgen/lippgen.d"
   ```

   in the `lippgen` script.
2. Check if the pre-configured port number 12349 of `lippgen` is free on your machine — if not, change it to something else in the line

   ```
   PORT = 12349
   ```

   Modify the command which opens a URL in a web browser; it is currently set to

   ```
   BROWSER_COMMAND = \
     "open %s -a \"Google Chrome\""
   ```

   which works on a Mac with Google Chrome installed. For Firefox on a Linux machine the proper command would be

   ```
   BROWSER_COMMAND = "firefox -new-tab %s"
   ```

4.2 Using LiPPGen

Assume you have a literate program in a file named `example.nw` (which is the standard file extension if you use `noweb`). Any other extension except `.html` is fine as well.

1. Mark the relevant part(s) of the literate programming source file by inserting two lines

   ```
   %%% BEGIN LITERATE TEACHING %%%
   ```

   and

   ```
   %%% END LITERATE TEACHING %%%
   ```

   (without any leading spaces) around each part that is to be included in the slides. (You can omit the last end marker; in that case LiPPGen will go on processing until the end of the document.)
2. Run `./lippgen` on the file, e. g. by issuing the command `./lippgen example.nw`; this produces a file `example.form.html` and opens it in the preconfigured browser.

   The default language for syntax highlighting is C. If you want Python instead, use `Python` as a second argument to `lippgen`. If you use a different language, modify the program.
3. In the browser, fill in the text input boxes next to the code chunks. You can leave input boxes empty if you want to create slides that only have code on them. Click *Send* at the end of the page.
4. Submitting will transfer the input boxes' contents to the program's built-in server, where the processing continues. Your entries in the fields will also be saved in `example.lip` so that it will be reused if you run `lippgen` on the same file again (the input boxes will already be filled with the entries from the last time). This step creates the final presentation file `example.html` and opens it in the browser.
5. Check the resulting slides and make changes if necessary (going back to step 3).
6. Give the lecture.

4.3 Generating extra pages

It's also possible to create extra presentation pages without code, but LiPPGen has no way of knowing that in advance. After you've initially created the HTML slides, you can edit the HTML file and insert

```
<div class="slide">
<h1>Slide Title</h1>
<ul>
  <li>...</li>
  <li>...</li>
  ...
</ul>
</div>
```

blocks between other `div` elements of class `slide`. This will disrupt the enumeration of slides (and as a consequence scrolling will not work in slides after the first manually included one). Thus, for post-processing of a manually modified HTML file, there's an extra tool called `lippgen-sanitize` that will renumber the slides.

However, modifying the HTML file this way still makes it harder to keep the original literate program and the presentation in sync; when your regenerate the slides with `lippgen`, you lose the slides which you have added manually. Future versions of LiPPGen may improve this procedure.

4.4 Adding keywords

Currently syntax highlighting knows only a few keywords which typically occur in C or Python programs. They are registered in the `C_KEYWORDS` and `P_KEYWORDS` variables, e. g.

```
C_KEYWORDS = ["uint ", "int ", "char ",
 "#define", "typedef", "struct", "return",
 "#include", "if ", "else "]
```

If you want to add your own keywords to the list (so that LiPPGen will highlight them), just append them to the appropriate variable.

A future version of LiPPGen might use the `highlight` program [12], or similar, to provide better highlighting.

4.5 Publishing a LiPPGen presentation

If you want to publish a LiPPGen presentation on a website, you will need to copy the HTML file and the automatically generated `lippgen.d` subdirectory to the web server. All files in that subdirectory are referenced via relative `"lippgen.d/..."` URLs, so the file should display properly without further ado.

5 An example

I've developed LiPPGen as part of my Ph.D. research which mainly consists of implementing Ulix [3], a new Unix-like operating system using literate programming. To test whether the literate programming approach is helpful in an operating systems class, I'm going to convert parts of the literate program into slides and use them during lectures; that first real-world test is scheduled for the winter term 2013/14 when I'll be giving a course called "Implementing Operating Systems with Literate Programming" at Nuremberg University of Applied Sciences (TH Nürnberg). When the course starts, slides will be available from the course website [1].

Figure 2 shows an excerpt from the signal handling chapter of the unpublished Ulix code which implements the `kill` system call. That part of the book contains four code chunks, and LiPPGen will convert them into four slides which may then be described.

When calling `lippgen`, a browser window displays the generated HTML form, as shown in Figure 3. You can then enter the slide contents and also provide metadata for the title slide (title, author, and organization). After clicking the *Send* button, the browser opens the final presentation file, shown in Figure 4.

6 License

The licensing for LiPPGen may look a little irritating, but since I've used other components, I need to follow their licenses. Thus, the modified S5 code is in the public domain, the NicEdit component is available under the MIT license, and the Python script `lippgen` is GPLv2-licensed.

To summarize this, you're basically free to do whatever you like with the package. If you modify and re-publish LiPPGen, you just have to be aware of the third party code's licenses.

7 Future work

I've also looked into the alternative presentation tool Prezi [11] which allows zooming into and out of presentation parts. There, the presentation is basically a big mind map.

Since using code chunks is somewhat similar to the Prezi approach, it would be interesting to have a tool which allows quick replacement of a chunk name with the chunk content (when clicking it).

I'm also planning to experiment with the abovementioned `highlight` program [12] since it makes no sense to invest time into developing a separate highlighting engine when similar code is available.

References

[1] Hans-Georg Eßer. Implementing Operating Systems With Literate Programming. Lecture slides, Nuremberg University of Applied Sciences, Winter term 2013/14. `http://ohm.hgesser.de/be-ws2013/`.

[2] Hans-Georg Eßer. LiPPGen. `http://hgesser.de/software/lippgen/`.

[3] Felix Freiling and Hans-Georg Eßer. ULIX. `http://www.ulixos.org/`.

[4] Brian Kirchoff. NicEdit Inline Content Editor, 2008. `http://nicedit.com/`.

[5] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[6] Donald E. Knuth. *TEX: The Program.* Addison Wesley Publishing Company, 1986.

[7] Literate programming link list. `http://www.literateprogramming.com/links.html`.

[8] Eric A. Meyer. S5: A Simple Standards-Based Slide Show System. `http://meyerweb.com/eric/tools/s5/`.

[9] Coji Morishita. M+ 1M font. `http://mplus-fonts.sourceforge.jp/mplus-outline-fonts/design/index-en.html`.

[10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[11] Prezi website. http://prezi.com/.
[12] Andre Simon. Highlight Manual. http://www.andre-simon.de/doku/highlight/en/highlight.html.
[13] Till Tantau et al. Beamer document class for LaTeX. http://ctan.org/pkg/beamer.

◇ Hans-Georg Eßer
Univ. Erlangen-Nürnberg
Lehrstuhl 1 für Informatik
Martensstraße 3
D-91058 Erlangen, Germany
h.g.esser (at) cs dot fau dot de
http://hgesser.de/

323a ⟨*kernel declarations* 59b⟩+≡ (61a) ◁322c 327a▷

```
void kill (int pid, int signo);
```

Uses kill.

Note that we do no checking in this function, kill can be called by the kernel itself (which may send any signal to any process), but it cannot be called directly by a process. Sending by a process requires using a system call, and the system call handler will check whether the process is allowed to send the signal to the target process before calling kill.

It is also classical for a process to send a signal to itself; that is what the raise function does. We will not implement it specifically inside the kernel, but in the user mode library: raise(sig) is the same as kill(getpid(),sig).

Here's the code for the system call handler:

323b ⟨*initialize syscalls* 94c⟩+≡ (64a) ◁313b

```
insert_syscall (__NR_kill, syscall_kill);
```

Uses insert_syscall and syscall_kill.

323c ⟨*syscall functions* 93c⟩+≡ (86d) ◁100f

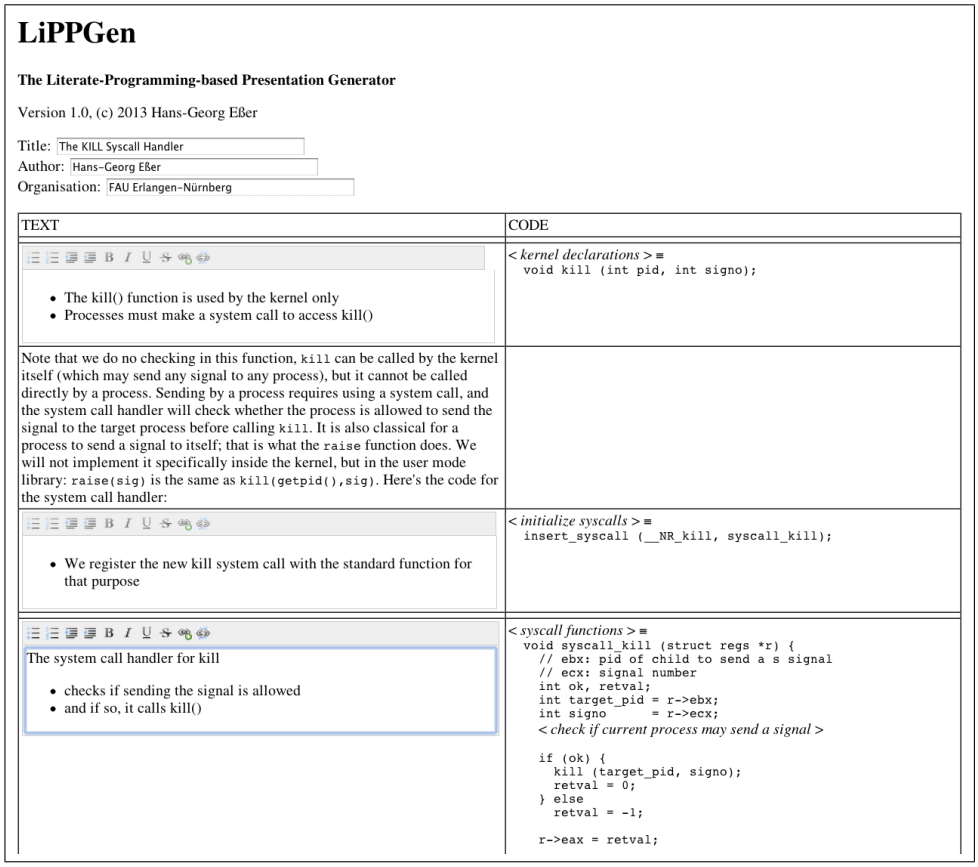
```
void syscall_kill (struct regs *r) {
  // ebx: pid of child to send a s signal
  // ecx: signal number
  int ok, retval;
  int target_pid = r->ebx;
  int signo      = r->ecx;
  ⟨check if current process may send a signal 323d⟩

  if (ok) {
    kill (target_pid, signo);
    retval = 0;
  } else
    retval = -1;

  r->eax = retval;

  ⟨run scheduler if this was a raise operation 324⟩
  return;
};
```

Uses kill and syscall_kill.

We only allow sending a signal if either the sender's owner has user ID 0 or if sender and recipient have the same owner:

323d ⟨*check if current process may send a signal* 323d⟩≡ (323c)

Figure 2: This excerpt from the literate program "Ulix" contains four code chunks.



LiPPGen

The Literate-Programming-based Presentation Generator

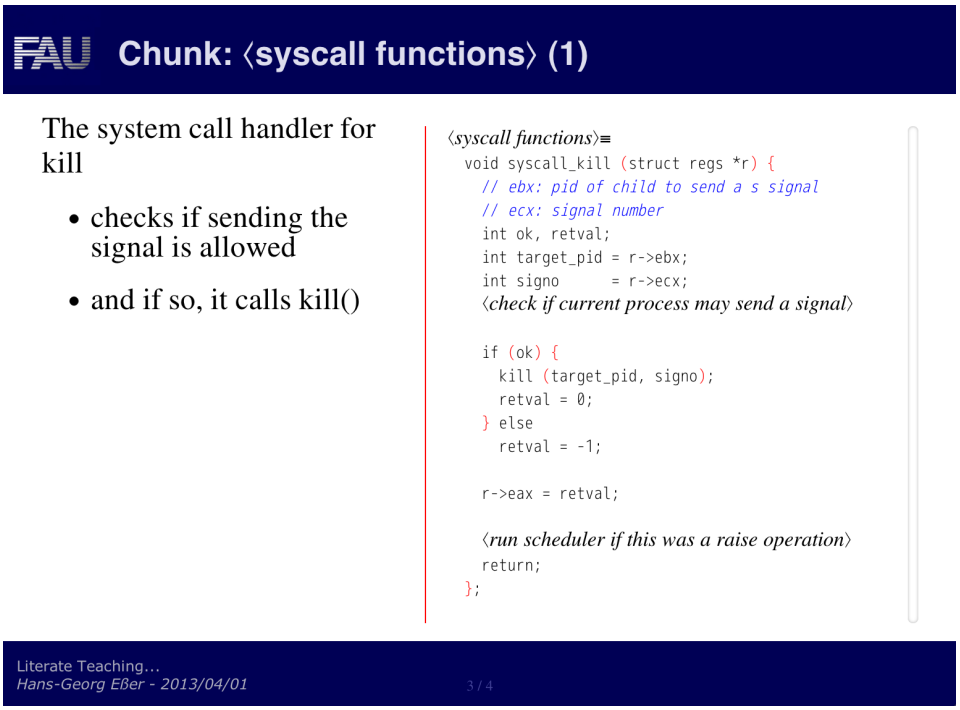
Version 1.0, (c) 2013 Hans-Georg Eßer

Title: The KILL Syscall Handler
Author: Hans-Georg Eßer
Organisation: FAU Erlangen-Nürnberg

TEXT	CODE
• The kill() function is used by the kernel only • Processes must make a system call to access kill()	< *kernel declarations* > ≡ `void kill (int pid, int signo);`
Note that we do no checking in this function, `kill` can be called by the kernel itself (which may send any signal to any process), but it cannot be called directly by a process. Sending by a process requires using a system call, and the system call handler will check whether the process is allowed to send the signal to the target process before calling `kill`. It is also classical for a process to send a signal to itself; that is what the `raise` function does. We will not implement it specifically inside the kernel, but in the user mode library: `raise(sig)` is the same as `kill(getpid(),sig)`. Here's the code for the system call handler:	
• We register the new kill system call with the standard function for that purpose	< *initialize syscalls* > ≡ `insert_syscall (__NR_kill, syscall_kill);`

The system call handler for kill
• checks if sending the signal is allowed
• and if so, it calls kill()
	< *syscall functions* > ≡
	`void syscall_kill (struct regs *r) {`
	`  // ebx: pid of child to send a s signal`
	`  // ecx: signal number`
	`  int ok, retval;`
	`  int target_pid = r->ebx;`
	`  int signo      = r->ecx;`
	`  < check if current process may send a signal >`
	`  if (ok) {`
	`  kill (target_pid, signo);`
	`  retval = 0;`
	`  } else`
	`  retval = -1;`
	`  r->eax = retval;`

Figure 3: The HTML form lets you enter content for the slides.



FAU Chunk: ⟨syscall functions⟩ (1)

The system call handler for kill

- checks if sending the signal is allowed
- and if so, it calls kill()

⟨*syscall functions*⟩≡

```
void syscall_kill (struct regs *r) {
  // ebx: pid of child to send a s signal
  // ecx: signal number
  int ok, retval;
  int target_pid = r->ebx;
  int signo      = r->ecx;
  ⟨check if current process may send a signal⟩

  if (ok) {
    kill (target_pid, signo);
    retval = 0;
  } else
    retval = -1;

  r->eax = retval;

  ⟨run scheduler if this was a raise operation⟩
  return;
};
```

Literate Teaching...
Hans-Georg Eßer - 2013/04/01 3 / 4

Figure 4: The HTML presentation file created by LiPPGen. The listing on the right hand side is scrollable.