pbook.el

Luke Gorrie, with modifications by Paul Khuong

January 31, 2007

Contents

1 Introduction

Have you ever printed out a program and read it on paper?

It is an interesting exercise to try with one of your own programs, one that you think is well-written. The first few times you will probably find that it's torture to try and read in a straight line. What seemed so nice in Emacs is riddled with glaring problems on paper.

How a program reads on paper may not be very important in itself, but there is wonderful upside to this. If you go through the program with a red pen and fix all the mind-bendingly obvious problems you see, what happens is that the program greatly improves – not just on paper, but also in Emacs!

This is a marvellously effective way to make programs better.

Let's explore the idea some more!

2 ‘pbook’

This program, `pbook`, is a tool for making readable programs by generating LaTeX’ified program listings. Its purpose is to help you improve your programs by making them read well on paper. It serves this end by generating pretty-looking PDF output for you to print out and attack with a red pen, and perhaps use the medium to trick your mind into seeking the clarity of a technical paper and bringing your prose-editing skills to bear on your source code.

`pbook` is aware of three things: headings, top-level comments, and code. Headings become LaTeX sections, and have entries in a table of contents. Top-level comments become plain text in a nice variable-width font. Other source code is listed as-is in a fixed-width font.

These different elements are distinguished in the source using maximally unobtrusive markup, which you can see at work in the `pbook.el` source code.

Read on to see the program and how it works.

3 Prelude

I have successfully tested this program with GNU Emacs versions 20.7 and 21.3, and with XEmacs version 21.5.

For some tiny luxuries and portability help we use the Common Lisp compatibility library:

```
0:  (require 'cl)
```

4 Emacs commands

A handful of Emacs commands make up the pbook user-interface. The most fundamental is to render a pbook-formatted Emacs buffer as LaTeX.

```
0:  (defun PBOOK-BUFFER ()
1:    "Generate LaTeX from the current (pbook-formatted) buffer.
2:  The resulting source is displayed in a buffer called *pbook*."
3:    (interactive)
4:    (pbook-process-buffer))
```

A very handy utility is to display a summary of the buffer’s structure and use it to jump to an appropriate section. I’ve always enjoyed being able to do this in texinfo-mode. Happily, pbook gets this for free using the `occur` function, which lists all lines in the buffer that match some regular expression.

```
0:  (defun PBOOK-SHOW-STRUCTURE ()
1:    "Display the pbook heading structure of the current buffer."
2:    (interactive)
3:    (occur pbook-heading-regexp))
```

To avoid a lot of mucking about in the shell there is also a command to generate and display a PDF file. This function is a quick hack to make experimentation easy.

```
(defun PBOOK-BUFFER-VIEW-PDF ()
  "Generate and display PDF from the current buffer.
The intermediate files are created in the standard temporary
directory."
  (interactive)
  (save-window-excursion
    (pbook-buffer))
  (with-current-buffer "*pbook*"
    (let ((texfile (pbook-tmpfile "pbook" "tex"))
          (pdffile (pbook-tmpfile "pbook" "pdf")))
      (write-region (point-min) (point-max) texfile)
      ;; Possibly there is a better way to ensure that LaTeX generates
      ;; the table of contents correctly than to run it more than
      ;; once, but I don't know one.
      (shell-command (format "\
  cd /tmp; latex %s && pdflatex %s && acroread %s &"
                             texfile texfile pdffile)))))

(defun PBOOK-TMPFILE (name extension)
  "Return the full path to a temporary file called NAME and with EXTENSION.
An appropriate directory is chosen and the PID of Emacs is inserted
before the extension."
  (format "%s%s-%S.%s"
          (if (boundp 'temporary-file-directory)
              temporary-file-directory
            ;; XEmacs does it this way instead:
            (temp-directory))
          name (emacs-pid) extension))
```

5 Configurable variables

These are variables that can be customized to affect pbook's behaviour. The default regular expressions assume Lisp-style comment characters, but they can be overridden with buffer-local bindings from hooks for other programming modes. The other variables that control formatting are best configured with Emacs's magic "file variables" (see down the very bottom for an example).

```
(defvar PBOOK-COMMENTARY-REGEXP "^;;;\\($\\|[^#]\\)"
  "Regular expression matching lines of high-level commentary.")

(defvar PBOOK-HEADING-REGEXP "^;;;\\(#+\\)"
  "Regular expression matching heading lines of chapters/sections/headings.")

(defvar PBOOK-HEADING-LEVEL-SUBEXP 1
  "The subexpression of `pbook-heading-regexp' whose length indicates nesting.")
```

```
(defvar PBOOK-INCLUDE-TOC t
  "When true include a table of contents.")

(defvar PBOOK-STYLE 'article
  "Style of output. Either article (small) or book (large).")

(defvar PBOOK-AUTHOR (user-full-name)
  "The name to use in the \author LaTeX command.")

(defvar PBOOK-CODE-PROLOGUE "\
\\vspace{1pc}
\\begin{adjustwidth}{0in}{-1.5in}
\\begin{flushleft}
"
  "Tex string to prepend to code listings")

(defvar PBOOK-CODE-EPILOGUE "\
\\end{flushleft}
\\end{adjustwidth}
\\vspace{1pc}
"
  "Tex string to append to code listings")

(defun PBOOK-AROUND-CODE-LINE (line-number total-lines)
  "returns a list of Tex strings '(prepend append)' to
surround the line in a code listing.
It receives, as its single argument, the line number (as an integer)."
  (labels ((repeat (string num)
             (if (<= num 0)
                 ""
               (concat string (repeat string (1- num))))))
    (if (looking-at "^[:space:]*$")
        (list "~" "\\\\")
      (let ((str (number-to-string line-number)))
        (list (concat "\\hspace{-.5in}\\texttt{"
                      (repeat "\\ " (max 1 (- 4 (length str))))
                      str
                      ":\\ \\ "
                      "}")
              (if (= line-number (- total-lines 1))
                  "\\\\"
                "\\nopagebreak[4]\\\\"))))))

(defvar PBOOK-DARK-COLORS '("black"))

(defvar PBOOK-FACE-LATEX-PROPERTIES '()
  "plist of latex properties for current face
 (only active while calling functions in `pbook-properties')")
```

```
58:  (defvar PBOOK-MONOCHROME nil
59:    "Force every color to be the specified color (list of rgb components)
60:  or t for standard black.")

62:  (defvar PBOOK-FACE-OVERRIDE '((font-lock-keyword-face :bold t)
63:                                (font-lock-builtin-face :bold t)
64:                                (font-lock-function-name-face
65:                                                        :sc t :tt :no)
66:                                (font-lock-variable-name-face
67:                                                        :sc t :tt :no)
68:                                (font-lock-warning-face :bold t)
69:                                (font-lock-comment-face :italic t :tt :no)
70:                                (font-lock-doc-face     :italic t :tt :no)
71:                                (font-lock-constant-face :italic :no))
72:    "alist of face -> plist")
```

```
(defvar PBOOK-PROPERTIES
  `((:color
     pbook-face-color
     ,(lambda (colors)
        (if (or (every (lambda (color)
                         (< color 0.01))
                       colors)
                (eq pbook-monochrome t))
            nil
          (let ((components (mapcar (lambda (color)
                                      (if (< color 0.01)
                                          "0"
                                        (number-to-string color)))
                                    (or pbook-monochrome
                                        colors))))
            (list (concat "\\textcolor[rgb]{"
                          (first components)
                          ", "
                          (second components)
                          ", "
                          (third components)
                          "}{")
                  "}")))))
    (:bold   face-bold-p
             ,(lambda (prop)
                (unless (eq prop :no)
                  (list "\\textbf{" "}"))))
    (:italic face-italic-p
             ,(lambda (prop)
                (unless (eq prop :no)
                  (list  "\\textit{" "}"))))
    (:tt     ,(lambda (face)
                t)
             ,(lambda (prop)
                (unless (eq prop :no)
                  (list "\\texttt{" "}"))))
    (:sc     ,(lambda (face)
                nil)
             ,(lambda (prop)
                (unless (eq prop :no)
                  (list "\\textsc{" "}")))))
  "alist of property -> (face-predicate prop->latex)
Where face-predicate = face -> property-value
      prop->latex    = prop-value -> (prepend append) — (prepend append)")
```

```
(defvar PBOOK-ESCAPING-REGEXPS '(("<" . "\\\\textless{}")
                                 (">" . "\\\\textgreater{}")
                                 ("\\\\" . "\\\\textbackslash{}")
                                 ("~" . "\\\\textasciitilde{}")
                                 ("\\^" . "\\\\textasciicircum{}")
                                 ("[ #%&$_{}]" . "\\\\\\&"))
  "alist of regexp -> replacement (passed to re-search-forward and replace-match)")

(defun PBOOK-FACE-COLOR (face)
  (let ((dark-bg-p (or (and (boundp 'face-background-mode)
                            (eq face-background-mode 'dark))
                       (member (face-background face)
                               pbook-dark-colors))))
    (and (face-foreground face)
         (let ((rgb-specs (mapcar (lambda (n)
                                    (/ n 65535.0))
                                  (color-values (face-foreground face)))))
           (and rgb-specs
                (if dark-bg-p
                    (let ((yuv-specs (apply 'pbook-rgb-yuv rgb-specs)))
                      (pbook-yuv-rgb (- 1 (first yuv-specs))
                                     (second yuv-specs)
                                     (third yuv-specs)))
                  rgb-specs))))))

(defun PBOOK-RGB-YUV (r g b)
  "As http://en.wikipedia.org/wiki/YUV"
  (let* ((y (+ (*  0.299    r)
               (*  0.587    g)
               (*  0.114    b)))
         (u (+ (* -0.168740 r)
               (* -0.331260 g)
               (*  0.500000 b)))
         (v (+ (*  0.500000 r)
               (* -0.418690 g)
               (* -0.081310 b))))
    (list y u v)))
```

```
157:  (defun PBOOK-YUV-RGB (y u v)
158:    "As http://en.wikipedia.org/wiki/YUV . Matrix fixed by mjp"
159:    (let ((r (+ y
160:
161:                (*  1.40200 v)))
162:          (g (+ y
163:                (* -0.34413 u)
164:                (* -0.71414 v)))
165:          (b (+ y
166:                (*  1.77200 u))))
167:      (mapcar (lambda (x)
168:                (cond ((< x 0) 0.0)
169:                      ((> x 1) 1.0)
170:                      (t       x)))
171:              (list r g b))))
```

6 Top-level logic

Here we have the top level of the program. Setting up, calling the formatting engine, piecing things together, and putting on the finishing touches.

The real work is done in a new buffer called *pbook*. First the source is copied into this buffer and from there it is massaged into shape.

Most of this is mundane, but there is one tricky part: the source buffer may have buffer-local values for some pbook settings, and we have to be careful or we'd lose them when switching into the *pbook* buffer. This is taken care of by moving the correct values of all the relevant customizable settings into new dynamic bindings.

```
(defun PBOOK-PROCESS-BUFFER ()
  "Generate pbook output for the current buffer
The output is put in the buffer *pbook* and displayed."
  (interactive)
  (unless pbook-monochrome
    (font-lock-default-fontify-buffer))
  (let ((buffer    (current-buffer))
        (beginning (pbook-tex-beginning))
        (ending    (pbook-tex-ending))
        (text      (buffer-string)))
    (with-current-buffer (get-buffer-create "*pbook*")
      ;; Setup,
      (pbook-inherit-buffer-locals buffer
                                   '(pbook-commentary-regexp
                                     pbook-heading-regexp
                                     pbook-style))
      (erase-buffer)
      (insert text)
      ;; Reformat as LaTeX,
      (pbook-preprocess)
      (pbook-format-buffer)
      ;; Insert header & footer.
      (goto-char (point-min))
      (insert beginning)
      (goto-char (point-max))
      (insert ending)
      (display-buffer (current-buffer)))))

(defun PBOOK-INHERIT-BUFFER-LOCALS (buffer variables)
  "Make buffer-local bindings of VARIABLES using the values in BUFFER."
  (dolist (v variables)
    (set (make-local-variable v)
         (with-current-buffer buffer (symbol-value v)))))

(defun PBOOK-PREPROCESS ()
  "Cleanup the buffer to prepare for formatting."
  (goto-char (point-min))
  ;; FIXME: Currently we just zap all pagebreak characters.
  (save-excursion
    (while (re-search-forward "\C-l" nil t)
      (replace-match "")))
  (unless (re-search-forward pbook-heading-regexp nil t)
    (error "File must have at least one heading."))
  (beginning-of-line)
  ;; Delete everything before the first heading.
  (delete-region (point-min) (point)))
```

```
47:  (defun PBOOK-TEX-BEGINNING ()
48:    "Return the beginning prelude for the LaTeX output."
49:    (format "\
50:  \\documentclass[notitlepage,a4paper]{%s}
51:  \\usepackage[nohead,nofoot]{geometry}
52:  \\usepackage{color}
53:  \\usepackage{bold-extra}
54:  \\usepackage{chngpage}
55:  \\title{%s}
56:  \\author{%s}
57:  \\begin{document}
58:  \\maketitle
59:  %s\n"
60:            (symbol-name pbook-style)
61:            (pbook-latex-escape-string (buffer-name))
62:            (pbook-latex-escape-string pbook-author)
63:            (if pbook-include-toc "\\tableofcontents" "")))

65:  (defun PBOOK-TEX-ENDING ()
66:    "Return the ending of the LaTeX output."
67:    "\\end{document}\n")
```

7 Escaping special characters

We have to escape characters that LaTeX treats specially. This is done based on the rules in the `Special Characters` node of the LaTeX2e info manual.

```
0:  (defun PBOOK-LATEX-ESCAPE-STRING (string)
1:    (with-temp-buffer
2:      (insert string)
3:      (pbook-latex-escape (point-min) (point-max))
4:      (buffer-string)))
```

```
(defun PBOOK-LATEX-ESCAPE (start end &optional space)
  "LaTeX-escape special characters in the region from START to END."
  (save-excursion
    (save-restriction
      (narrow-to-region start end)
      (goto-char (point-min))
      (save-excursion
        (while (re-search-forward "\\\\" nil t)
          (replace-match "$\\backslash$" nil t)))
      (save-excursion
        (while (re-search-forward (if space
                                      "\\([#%&~$_^{} ]\\)"
                                    "\\([#%&~$_^{}]\\)")
                                  nil t)
          ;; Don't re-escape our escaped backslashes.
          (if (and (equal (char-before) ?\$)
                   (looking-at (regexp-quote "\\backslash$")))
              (goto-char (match-end 0))
            (replace-match "\\\\\\1")))))))
```

8 Processing engine

The main loop scans through the source buffer piece by piece and converts each one to LaTeX as it goes. There are three sorts of pieces: headings, top-level commentary, and code.

This loop recognises what type of piece is at the point and then calls the appropriate subroutine. The subroutines are responsible for determining where their piece finishes and for advancing the point beyond the region they have formatted.

```
(defun PBOOK-FORMAT-BUFFER ()
  (while (not (eobp))
    (if (looking-at "^\\s *$")
        ;; Skip blank lines.
        (forward-line)
      (cond ((looking-at pbook-heading-regexp)
             (pbook-do-heading))
            ((looking-at pbook-commentary-regexp)
             (pbook-do-commentary))
            (t
             (pbook-do-code))))))
```

8.1 Heading formatting

Each heading line is converted to a LaTeX sectioning command. The heading text is escaped.

```
(defun PBOOK-DO-HEADING ()
  ;; NB: 'looking-at' sets the Emacs match data (for match-string, etc)
  (assert (looking-at pbook-heading-regexp))
  (let ((depth (length (match-string-no-properties pbook-heading-level-subexp))))
    ;; Strip off the comment characters and whitespace.
    (replace-match "")
    (when (looking-at "\\s +")
      (replace-match ""))
    (pbook-latex-escape (line-beginning-position) (line-end-position))
    (wrap-line (format "\\%s{" (pbook-nth-sectioning-command depth))
               "}"))
  (forward-line))

(defun WRAP-LINE (prefix suffix)
  "Insert PREFIX at the start of the current line and SUFFIX at the end."
  (save-excursion
    (goto-char (line-beginning-position))
    (insert prefix)
    (goto-char (line-end-position))
    (insert suffix)))
```

LaTeX has different sectioning commands for articles and books, so we have to choose from the right set. These variables define the sets in order of nesting – the first element is top-level, etc.

```
(defconst PBOOK-ARTICLE-SECTIONING-COMMANDS
  '("section" "subsection" "subsubsection")
  "LaTeX commands for sectioning articles.")

(defconst PBOOK-BOOK-SECTIONING-COMMANDS
  (cons "chapter" pbook-article-sectioning-commands)
  "LaTeX commands for sectioning books.")

(defun PBOOK-NTH-SECTIONING-COMMAND (n)
  "Return the sectioning command for nesting level N (top-level is 1)."
  (let ((commands (ecase pbook-style
                    (article pbook-article-sectioning-commands)
                    (book    pbook-book-sectioning-commands))))
    (nth (min (1- n) (1- (length commands))) commands)))
```

8.2 Commentary formatting

Top-level commentary is stripped of its comment characters and we escape all characters that LaTeX treats specially.

```
(defun PBOOK-DO-COMMENTARY ()
  "Format one or more lines of commentary into LaTeX."
  (assert (looking-at pbook-commentary-regexp))
  (let ((start (point)))
    ;; Strip off comment characters line-by-line until end of section.
    (while (or (looking-at pbook-commentary-regexp)
               (and (looking-at "^\\s *$")
                    (not (eobp))))
      (replace-match "")
      (delete-horizontal-space)
      (forward-line))
    (save-excursion
      (pbook-latex-escape start (point))
      (pbook-pretty-commentary start (point)))))
```

These functions define a simple Wiki-like markup language for basic formatting.

```
(defun PBOOK-PRETTY-COMMENTARY (start end)
  "Make commentary prettier."
  (save-restriction
    (narrow-to-region start end)
    (goto-char (point-min))
    (save-excursion (pbook-pretty-tt))
    (save-excursion (pbook-pretty-doublequotes))))

(defun PBOOK-PRETTY-TT ()
  "Format `single quoted' text with a typewriter font."
  (while (re-search-forward "`\\([^']*\\)'" nil t)
    (replace-match "{\\\\tt \\1}" t)))

(defun PBOOK-PRETTY-DOUBLEQUOTES ()
  "Format \"double quoted\" text with ``double single quotes''."
  (while (re-search-forward "\"\\([^\"]*\\)\"" nil t)
    (replace-match "``\\1''")))
```

8.3 Source code formatting

Source text is rendered in the `verbatim` environment.

```
(defun PBOOK-DO-CODE ()
  (assert (and (not (looking-at pbook-commentary-regexp))
               (not (looking-at pbook-heading-regexp))))
  (let ((start (point))
        (end   (progn
                 (pbook-goto-end-of-code)
                 (point))))
    (save-restriction
      (narrow-to-region start end)
      (pbook-convert-tabs-to-spaces start end)
      ;; delete trailing newlines and spaces
      (goto-char (point-max))
      (while (or (equal (char-syntax (char-before)) " ")
                 (bolp))
        (delete-char -1))
      (pbook-escape-code start (point-max) (count-lines start (point-max)))
      (goto-char (point-min))
      (insert pbook-code-prologue)
      (goto-char (point-max))
      (insert "\n" pbook-code-epilogue "\n"))))

(defun PBOOK-GOTO-END-OF-CODE ()
  "Goto the end of the current section of code."
  (if (re-search-forward (format "\\(%s\\)\\|\\(%s\\)"
                                 pbook-heading-regexp
                                 pbook-commentary-regexp)
                         nil t)
      (beginning-of-line)
    (goto-char (point-max))))

(defun PBOOK-CONVERT-TABS-TO-SPACES (start end)
  "Replace tab characters with spaces."
  (save-excursion
    (save-restriction
      (narrow-to-region start end)
      (untabify start end))))
```

The escaping rules for verbatim environments are unclear to me. It looks like the only thing that needs escaping is `\end{verbatim}`, but I don't know of an escape mechanism that works (\ is taken literally). Since most programs (apart from this one..) won't contain that string I have made a kludge to fudge it by inserting an underscore.

```
(defun PBOOK-ESCAPE-CODE (start end num-lines)
  "Escape verbatim source code for LaTeX."
  (save-excursion
    (save-restriction
      (narrow-to-region start end)
      (goto-char (point-min))
      (let ((cur-line 0))
        (while (< cur-line num-lines)
          (pbook-escape-line (line-beginning-position) (line-end-position)
                             cur-line num-lines)
          (incf cur-line)
          (beginning-of-line 2))))))

(defun PBOOK-GET-INHERITS (face)
  (let ((faces (cond ((eq face 'unspecified) nil)
                     ((listp face)          face)
                     (t                     (list face)))))
    (mapcan (lambda (face)
              (let ((inherits (face-attribute face :inherit)))
                (if (and inherits
                         (not (eq inherits 'unspecified)))
                  (cons face (pbook-get-inherits inherits))
                  (list face))))
            faces)))

(defun PBOOK-TRANSLATE-FACE-PROPERTIES (face props)
  (setq props (append props
                      (copy-list (cdr (assoc face pbook-face-override)))))
  (dolist (defn pbook-properties)
    (let ((prop-name (first defn))
          (predicate (second defn)))
      (unless (plist-get props prop-name)
        (let ((value (funcall predicate face)))
          (when value
            (setq props (plist-put props prop-name value)))))))
  props)

(defun PBOOK-FACE-PROPERTIES (face)
  (let ((faces (append (pbook-get-inherits face)
                       '(default)))
        (props nil))
    (dolist (face faces props)
      (setq props (pbook-translate-face-properties face props)))))
```

```
(defun PBOOK-PROPERTIES-LATEX-STRINGS (plist)
  (let ((prepend nil)
        (append  nil)
        (pbook-face-latex-properties plist))
    (dolist (property pbook-properties (list (apply 'concat
                                                    (reverse prepend))
                                             (apply 'concat append)))
      (let* ((name         (first property))
             (transformer  (third property))
             (foundp (plist-member plist name))
             (prop   (plist-get plist name)))
        (when foundp
          (let ((wrap (if (listp transformer)
                          (and (not (eq prop :no))
                               prop
                               transformer)
                        (funcall transformer prop))))
            (when wrap
              (push (first wrap) prepend)
              (push (second wrap) append)))))))))

(defun PBOOK-ESCAPE-LINE (start end line-number total-lines)
  (save-excursion
    (save-restriction
      (narrow-to-region start end)
      (goto-char start)
      (let ((substr-beg (point-marker))
            (substr-end (point-marker))
            (wrap       (pbook-around-code-line line-number total-lines)))
        (move-marker substr-end
                     (next-char-property-change (marker-position substr-beg)))
        (set-marker-insertion-type substr-beg t)
        (set-marker-insertion-type substr-end t)
        (while (not (equal substr-beg substr-end))
          (goto-char (marker-position substr-beg))
          (let ((wrap (pbook-properties-latex-strings
                       (pbook-face-properties
                        (get-char-property (marker-position substr-beg)
                                           'face)))))
            (insert (first wrap))
            (pbook-escape-code-substring (marker-position substr-beg)
                                         (marker-position substr-end))
            (goto-char (marker-position substr-end))
            (insert (second wrap))
```

```
89:              (move-marker substr-beg (marker-position substr-end))
90:              (move-marker substr-end
91:                           (next-char-property-change
92:                            (marker-position substr-beg)))))
93:          (wrap-line (first wrap)
94:                     (second wrap))))))

96:  (defun PBOOK-ESCAPE-CODE-SUBSTRING (start end)
97:    (when pbook-escaping-regexps
98:      (let ((scan-regexp (apply 'concat
99:                                (car (first pbook-escaping-regexps))
100:                               (mapcan (lambda (entry)
101:                                         (list "\\|"
102:                                               (car entry)))
103:                                       (rest pbook-escaping-regexps)))))
104:       (save-excursion
105:         (save-restriction
106:           (narrow-to-region start end)
107:           (goto-char start)
108:           (while (re-search-forward scan-regexp
109:                                     nil t)
110:             (goto-char (match-beginning 0))
111:             (dolist (entry pbook-escaping-regexps)
112:               (let ((test (car entry))
113:                     (replace (cdr entry)))
114:                 (when (looking-at test)
115:                   (replace-match replace)))))))))
```

9 Prologue and file variables

```
0:  (provide 'pbook)
```

We use Emacs's magic `file variables` to make sure pbook is formatted how it should be:

```
0:  ;; Local Variables:
1:  ;; pbook-author: "Luke Gorrie, with modifications by Paul Khuong"
2:  ;; pbook-use-toc: t
3:  ;; pbook-style:  article
4:  ;; End:
```