Contents

1 Startup

1.1 talcum.el

```
(defconst talcum-version "0.5.0")

;; Path setup.
;;
;; Technically, all that we want is all the .el files where we can
;; find them. This used to be done via talcum-path; it happens
;; automatically if Talcum is installed to the magic site-lisp
;; directories; and if at least this file is in the load-path already,
;; we will just assume that
;;   1. main Talcum files are in the same directory
;;   2. TREXes live in the subdir /trex/
;; since they fall out the tar.gz that way.

(let* ((thisfile (locate-library "talcum")))
  (cond
   ((and thisfile (not (boundp 'talcum-path)))
    ;; We are in the load-path already, and we need not honour
    ;; talcum-path.
    (setq talcum-path (file-name-directory thisfile))
    ;; If we are in the magic trees, we are done.
    (if (not (string-match "/site-lisp/" thisfile))
        ;; Otherwise, add the trex subdir.
        (add-to-list 'load-path (concat talcum-path "/trex/"))))
   (t
    ;; OK, we were called with absolute path the old-fashioned way.
    ;; Unfortunately, there seems to be no way of getting our position
    ;; while we're being loaded.
    (when (not (boundp 'talcum-path))
      ;; Yes well, what am I supposed to do?
      (setq talcum-path "/home/ulmi/Projekte/Emacs/Talcum/talcum/src/")
      (message "Warning: Could not figure out talcum-path, using %s" talcum-path))
    ;; Anyway, now we're ready.
    (add-to-list 'load-path talcum-path)
    (add-to-list 'load-path (concat talcum-path "/trex/")))))

;; Nothing works without these three.
(require 'talcum-tools)
(require 'talcum-compat)
(require 'talcum-custom)

(talcum-namespace
 (~~sourcefile-versionstamp "$RCSfile: talcum.el,v $" "$Revision: 2.11 $"))

(autoload 'talcum-add-render-feature
  "talcum-render" 'interactive)
```

Customize-Setup.

```
(defgroup talcum
  nil ;; members
```

```
  "A mixed bag of things that make LaTeX easier."
  :group 'tex)

(let ((unless-forbidden
       (lambda (flag sym)
         (if (and (boundp flag) (null (symbol-value flag)))
             nil
           (list sym)))))
  ;; We used to have talcum-use-foo-flags here. That's unfortunate,
  ;; optimally, we don't want to change this file when a new sublibrary
  ;; is added. Now, we have a list of features that will be required.
  (defcustom talcum-desired-features
    ;; append hack here: if those were customized to nil, obey that.
    (append
     (funcall unless-forbidden 'talcum-use-render-flag 'talcum-render)
     (funcall unless-forbidden 'talcum-use-prod-flag 'talcum-prod)
     (funcall unless-forbidden 'talcum-use-newcmd-flag 'talcum-newcmd)
     '(talcum-pikeman talcum-tabdance))
    "Features that should be loaded when Talcum is started"
    :group 'talcum
    :type '(repeat symbol) ;; fix this for something clickable
    ))

;; Two hooks provided for the proper work-doing code. We could do
;; without the -enter hook, but it's here for symmetry. Since we
;; dabble in idle-timers etc. pp., we want to provide a way to disable
;; them.
(defvar talcum-enter-mode-hook nil)
(defvar talcum-leave-mode-hook nil)
```

Settings of keys.

```
(define-prefix-command 'talcum-keymap)

;;
;; Main entry point for all things talcum
;;
(defconst talcum-lighter " Talcum")

(defcustom talcum-mode-chord (vector ?\C-c ?#)
  "The key combination prefix for all Talcum commands."
  :group 'talcum
  :type 'sexp)
```

###autoload

```
(define-minor-mode talcum-mode
  "A minor mode to enhance both LaTeX- and latex-mode."
  nil                                   ; initially off
  talcum-lighter                ; mode indicator
  (list                                 ; keymap
   (cons talcum-mode-chord 'talcum-keymap))
  ;; init body

  ;; Hmmm... I think this should work. The docs are very unclear on
```

```
  ;; whether the autoload cookie will load the whole file. Let's make
  ;; sure, because without namespaces, we shan't get far.
  (require 'talcum)
  (require 'talcum-menu)

  (when talcum-mode ;; activating!
    (talcum-namespace
     (cond
      ((talcum-latex-p) (~~debug 1 "Classic latex mode, ok."))
      ((talcum-auctex-p) (~~debug 1 "AucTeX LaTeX mode, ok."))
      (t (~~debug 9 (concat "Unsupported major mode \"%S\". "
                            "Things might go wrong.")
                    major-mode)))

     ;; more activation stuff goes here
     (dolist (feat talcum-desired-features)
       (~~debug 1 "Requiring %s" feat)
       (require feat))
     (require 'talcum-bugreport)
     (run-hooks 'talcum-enter-mode-hook)
     (talcum-menu-regenerate)
     t))
  (unless talcum-mode ;; deactivating!
    (run-hooks 'talcum-leave-mode-hook)
    t)
  )

(provide 'talcum)
```

talcum.el ends here.

2 If things go wrong...

2.1 talcum-bugreport.el

Bug reporting.

It's difficult for users to know what's important and what's not. Also, I tend to give out an erroneous email address.

```

(require 'talcum-tools)
(require 'talcum-menu)

(talcum-namespace
 (~~sourcefile-versionstamp "$RCSFile:" "$Revision:"))

(defun talcum-report-bug (subj)
  (interactive "sOne-line description: ")
  (compose-mail "ulmi@users.sarovar.org"
                (concat "[Talcum "
                        talcum-version
                        " Bug] "
```

```
                    subj) nil t)
  (insert "

Please describe your problem here. ulmi speaks English, German and
very little French. In many cases (e.g. display bugs), it is helpful
if a small sample file is included on which the bug is reproducible.

"

;; Please read and acknowledge the following legalese:

;; The bug report includes information about your system (most
;; importantly, operating system and Emacs version, and whether you use
;; AucTeX). This information is included in plain text, at the end of
;; this report.

;; You give permission for this mail to be published and incorporated,
;; whole or in parts, free of charge and obligation to the maintainer, in
;; future releases of Talcum (e.g. if you propose a fix) and/or on the
;; Talcum homepage (probably in the Bugs and Patches sections). The
;; maintainer will give credit where it's due, unless he forgets to (in
;; which case, please submit a bug about forgetting to credit).

"\nThank you for your help!\n      Ulrich"

"\nInternal information:\n")
  (mapc
   #'(lambda (sym)
      (insert
       (concat (symbol-name sym) "'s value: "
             (if (boundp sym)
                (format "%s" (symbol-value sym))
              "(void)")
             "\n")))
   '(
     emacs-version
     system-type
     system-configuration
     AucTeX-version
     talcum-version
     talcum--versions
     talcum-path
     talcum-desired-features
     ))
  t)

(defalias 'talcum-curse-at-programmer 'talcum-report-bug)

(defun talcum-bug-generate-menu ()
  (list "--"
        '["Report a bug" talcum-curse-at-programmer]))

(add-to-list 'talcum-menu-functions
```

```
          #'talcum-bug-generate-menu
          :at-end)

(provide 'talcum-bugreport)
```

3 In command of your commands

3.1 talcum-newcmd.el

$Id: talcum-newcmd.el,v 2.15 2005/08/27 12:41:08 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```

(require 'talcum-tools)
(require 'talcum-compat)
(require 'talcum-menu)


(talcum-namespace
 (~~sourcefile-versionstamp "$RCSfile: talcum-newcmd.el,v $" "$Revision: 2.15 $")

 (defun talcum-newcmd-generate-menu ()
  (list "--"
   '["Insert \\newcommand in preamble" talcum-insert-newcommand]
   '["Insert \\newenvironment in preamble" talcum-insert-newenvironment]
   '["Insert \\usepackage in preamble" talcum-insert-usepackage]))

 (add-to-list 'talcum-menu-functions #'talcum-newcmd-generate-menu)

 (defvar talcum-newfoo-before-insertion-hooks nil)
 (defvar talcum-newfoo-after-insertion-hooks nil)

 (defun ~~newfoo-find-suitable-place ()
  (let* ((talcum-definitions-file talcum-definitions-file)
        (goodbuffer
         ;; 2005AUG25: jumbling the order.
         ;; Doing functionp earlier obliterates the need for
         ;; :latexmode and :auctex flags -- auctex does not
         ;; provide a tex-main-file.
         (cond
          ((stringp talcum-definitions-file)
           talcum-definitions-file)
          ((functionp talcum-definitions-file)
           (funcall talcum-definitions-file))
          ((and ;(equal talcum-definitions-file ':latexmode)
            (boundp 'tex-main-file)
            (stringp tex-main-file))
           tex-main-file)
          ((and ;(equal talcum-definitions-file ':auctex)
```

```
          (talcum-auctex-p))
         (TeX-active-master t))
        (t (current-buffer))
        )))
  (when (stringp goodbuffer)
    (switch-to-buffer (find-file-noselect goodbuffer) 'secretly))
  (if buffer-read-only
      (vc-toggle-read-only))
  ;;(switch-to-buffer goodbuffer 'secretly)
  (~~newfoo-find-suitable-position)))

(defun ~~newfoo-find-suitable-position (&optional type)
  ;; This is hopeless. What _is_ a good position?
  ;;
  ;; Optimally, we would have all usepackages first, but their setup
  ;;commands (\hypersetup et al.) just after them, and all commands
  ;;after that.
  (goto-char (point-min))
  (search-forward "\\begin{document}" nil t)
  (forward-line 0)
  ;; Now just before begin{document}.
  ;; Always skip over author, title, date, etc.
  (dolist (textinfo talcum-textinfo-commands)
    (search-backward textinfo nil t))
  (when (eq type :package)
    ;; This misbehaves if no usepackage is present, but other
    ;; commands. Oh c'mon, that's bound to be rare.
    (when (search-backward "\\usepackage" nil t)
      (forward-char (length "\\usepackage"))
      (when (looking-at "[[:space:]]*\\[")
        ;; skip package args
        (skip-syntax-forward " ")
        (~~fwd-arg))
      (~~fwd-arg)
      (when (looking-at "[[:space:]]*\\[")
        ;; skip package release date
        (skip-syntax-forward " ")
        (~~fwd-arg))
      (forward-line 1))
    t)
  t)

(defmacro ~~at-suitable-place (&rest cmds)
  `(save-excursion
     (save-window-excursion
       (~~newfoo-find-suitable-place)
       ,@cmds)))


(defun talcum-excurse-to-preamble ()
  (interactive)
  (~~at-suitable-place
   (message "Press M-C-c when finished.")
```

```
    (recursive-edit)
    (save-buffer)))

 (defun talcum-insert-newcommand (pfx macname)
   "Jump to a suitable point and generate a skeleton for a LaTeX macro.
Where to jump is determined by ltx-newfoo-find-suitable-place.
With a prefix argument, assume that many parameters (0 otherwise).
With transient-mark-mode and active region, use that region as body."
   (interactive "P\nsName of macro: \\")

   (let* ((argnumber
         (cond
          ((null pfx) 0)
          ((integerp pfx) pfx)
          ((listp pfx) (car pfx)) ; empty list caught above
          ((eq pfx '-) -1)))       ; whoopsie! Think of something.
        (beg (and transient-mark-mode mark-active (region-beginning)))
        (end (and transient-mark-mode mark-active (region-end)))
        (body (if beg (delete-and-extract-region beg end)
              "%\n")))

     (when beg
      (deactivate-mark)
      (insert (format "\\%s%s"
                  macname
                  (if (and pfx (= 0 argnumber))
                      "\\ "
                    (apply 'concat
                         (make-list argnumber "{}")))))
      t)
     (~~at-suitable-place
      (run-hook-with-args 'talcum-newfoo-before-insertion-hooks
                    ':macro macname argnumber)
      (insert (format
             "\\newcommand\\%s%s{%%\n\n}\n\n"
             macname
             (if (= 0 argnumber) ""
               (format "[%s]" argnumber))
             body))
      (forward-line -3)
      (message "Press M-C-c when finished.")
      (recursive-edit)
      (run-hook-with-args 'talcum-newfoo-after-insertion-hooks
                    ':macro macname argnumber)
      (save-buffer)
      t)
     t))

 (defun talcum-insert-newenvironment (pfx envname)
   "Jump to a suitable point and generate a skeleton for a LaTeX environment.
Where to jump is determined by ltx-newfoo-find-suitable-place. The
prefix, if any, specifies the number of arguments."
   (interactive "P\nsName of environment: ")
```

```
  (let* ((argnumber
          (cond
           ((null pfx) 0)
           ((integerp pfx) pfx)
           ((listp pfx) (car pfx)) ; empty list caught above
           ((eq pfx '-) -1))))        ;  whoopsie! Think of something.

    (~~at-suitable-place
     (run-hook-with-args 'talcum-newfoo-before-insertion-hooks
                         ':environment envname argnumber)
     (insert (format "\\newenvironment{%s}%s{%%\n%%\n}{%%\n%%\n}\n\n"
                     envname
                     (if (= 0 argnumber) ""
                       (format "[%s]" argnumber))))
     (forward-line -5)
     (message "Press M-C-c when finished.")
     (recursive-edit)
     (run-hook-with-args 'talcum-newfoo-after-insertion-hooks
                         ':environment envname argnumber)
     (save-buffer))))


(defun talcum-insert-usepackage (package &optional options)
  (interactive "sPackage name: \n")
  (unless options
    (setq options
          (read-from-minibuffer
           "Options: "
           (cdr-safe (assoc package talcum-package-option-hints)))))
  (~~at-suitable-place
   (~~newfoo-find-suitable-position :package)
   (insert "\\usepackage"
           (if (< 0 (length options))
               (concat "[" options "]")
             "")
           "{" package "}\n")
   (sit-for 1.0)))

) ;;namespace
```

Init and deinit functions, for completeness.

```
(defun talcum-newcmd-init ()
  t)
(defun talcum-newcmd-exit ()
  t)
```

Key bindings

```
(define-key talcum-keymap "c" 'talcum-insert-newcommand) ; c for command
(define-key talcum-keymap "e" 'talcum-insert-newenvironment) ; e for envir
(define-key talcum-keymap "p" 'talcum-insert-usepackage) ; p for package
```

Customizations

```
394
395 (defgroup talcum-newcmd nil "Inserting new commands" :group 'talcum)
396
397  (defcustom talcum-textinfo-commands
398    '(;; LaTeX2e classes and everywhere:
399      "\\author" "\\date" "\\ title "
400      ;; KOMA-Script:
401      "\\publishers" "\\subject" "\\ titlehead "
402      "\\dedication" "\\uppertitleback" "\\lowertitleback" "\\ extratitle "
403      ;; beamer:
404      "\\ subtitle " "\\ institute " "\\subject" "\\keywords" "\\titlegraphic"
405      )
406    "Commands that are technically in the preamble, but should not be
407 seperated from begin{document}."
408    :group 'talcum-newcmd
409    :type '(repeat :tag "Macro name" string))
410
411 (defcustom talcum-use-newcmd-flag t
412   "*Enable the whole newcommand subsystem."
413   :group 'talcum
414   :type 'boolean)
415
416
417 (defcustom talcum-definitions-file nil
418   "Where Talcum will put new definitions (newcommands, newenvs)."
419   :group 'talcum-newcmd
420   :tag "Talcum definitions file"
421   :type '(choice
422         (const :tag "the current buffer" nil)
423         (const :tag "LaTeX-mode's master file (deprecated)" ':latexmode)
424         (const :tag "AucTeX' master file (deprecated)" ':auctex)
425         (string :tag "filename" :value "preamble.ltx")
426         (function :tag "returned by function" :value (lambda () "preamble.tex"))))
427
428
429
430 (defcustom talcum-package-option-hints
431   '(("inputenc" . "latin9")
432     ("fontenc" . "T1")
433     ("helvet" . "scaled"))
434   "Talcum can suggest options for packages."
435   :group 'talcum-newcmd
436   :tag "Package option suggestions"
437   :type '(repeat
438         (cons (string :tag "package name" :value "package")
439               (string :tag "option string" :value ""))))
440
441
442 (add-hook 'talcum-enter-mode-hook #'talcum-newcmd-init)
443 (add-hook 'talcum-leave-mode-hook #'talcum-newcmd-exit)
444
445 (provide 'talcum-newcmd)
```

talcum-newcmd.el ends here.

4 Better completion for a better tomorrow

4.1 talcum-tabdance.el

Tabdance: Enhanced Minibuffer Functionality.

We define an additional key in minibuffer maps to switch between different completion sets. The idea is that these sets offer increasing numbers of choices based on user behaviour: the first level will only offer elements from the history (so now "eq" will complete to "equation" if you never use eqnarray), the second level would maybe offer some predefined set as well, a third level could start parsing included packages for command definitions.

Currently, we use the first two steps: first, history, then, the original list, by setting talcum-tabdance-auto-prefer-history to a non-nil value. LaTeX mode's insert-block thing is redefined to use the history and thus benefit from this.

Known deficiency: We are currently limited because of a wart in GNU Emacs: read-file-name, read-buffer et al. don't honor advice to completing-read, hence, tabdance won't work with them. (This includes everything that is called by interactive codes, I think.)

```
449
450 (require 'talcum-tools)
451 (require 'talcum-compat)
452 (require 'talcum-custom)
453 (require 'talcum-menu)
454
455 (talcum-namespace
456  (~~sourcefile-versionstamp "$RCSfile: talcum-tabdance.el,v $"  "$Revision: 1.7 $"))
457
458 (defgroup talcum-tabdance
459   nil
460   "Improved minibuffer completion"
461   :group 'talcum)
462
463 (defcustom talcum-use-tabdance-flag nil
464   "*Set to non-nil to use tabdance."
465   :group 'talcum-tabdance
466   :type 'boolean)
467
468 (defcustom talcum-tabdance-key nil
469   "The key that will switch completion levels. Any value that is
470 suitable as a second argument to define-key. You may find it
471 convenient to change this by calling talcum-tabdance-query-key
472 instead."
473   :group 'talcum-tabdance
474   :type 'sexp)
```

Getter here, it is referenced in the docstring above.

```
476 (defun talcum-tabdance-query-key (chord)
477   (interactive "KKey sequence to get more completions in minibuffer: ")
```

```
  (setq talcum-tabdance-key chord))

(defcustom talcum-tabdance-auto-prefer-history t
  "*Make a two-step tabdance out of history and completion list
automagically, in all applications that use completing-read"
  :group 'talcum-tabdance
  :type 'boolean)

(defcustom talcum-tabdance-auto-escalate t
  "*If no completion exists at the current completion level,
automatically try a more choiceful level."
  :group 'talcum-tabdance
  :type 'boolean)
```

In intermediate CVS versions, this was not documented and checked with ~~value-safe. This will probably become obsolete if the default tex-latex-block becomes history-aware itself, unless by then they've completely gone over the border with asking for parameters etc.

```
(defcustom talcum-tabdance-no-hijacking t
  "Do not replace LaTeX-mode's tex-latex-block with a history-enabled
variant."
  :group 'talcum-tabdance
  :type 'boolean)

(talcum-namespace

 (defun talcum-tabdance-generate-menu ()
  (list "--"
       (~~menu-flag-toggle talcum-use-tabdance-flag
                           "Improved Minibuffer")
```

Oh well, I think they're stable enough. Use the customization.

```
;;          (~~menu-flag-toggle talcum-tabdance-auto-prefer-history
;;                              "Automatic History")
;;          (~~menu-flag-toggle talcum-tabdance-auto-escalate
;;                              "Automatic Escalation")
       ))

 (add-to-list 'talcum-menu-functions #'talcum-tabdance-generate-menu)


 (defun talcum-tabdance-escalate ()
   "Go to the next more choicy level of completion, if there is one.
Return a non-nil value if that is a real increase."
  (interactive)
  (when (boundp 'tabdance-level)
    (unless (= tabdance-level (~~pseudolength tabdance-choices))
      (if (interactive-p)
          (message "completion level %s" (1+ tabdance-level)))
      (setq tabdance-level (1+ tabdance-level)))))

 (defun talcum-tabdance-descalate ()
   "Go to the previous, more restricted level of completion if there is one.
Return a non-nil value if that is a real decrease"
```

```
  (interactive)
  (when (boundp 'tabdance-level)
   (unless (= 1 tabdance-level)
    (if (interactive-p)
       (message "completion level %s" tabdance-level))
    (setq tabdance-level (1- tabdance-level)))))
```

This is the one the user will want to call. Question: should the directions be inverted? With automatic escalation, maybe going back down will be more useful by default.

```
 (defun talcum-tabdance-scalate (direction)
  "Without prefix argument, go to a more choiceful completion level,
with prefix argument, go back one level."
  (interactive "P")
```

We call-interactively because we want the message.

```
  (call-interactively
   (if direction
      #'talcum-tabdance-descalate
     #'talcum-tabdance-escalate)))
```

Remove doubles from a list. This is likely to be phased out at some point, because it looks like GNU Emacs 22+ will have an option to remove doubles on its own. One way or another, this is not critical, but a completion window that lists equation five times looks... strange.

```
 (defun ~~sparsed (xs)
  (let (ys)
   (dolist (x xs ys)
    (add-to-list 'ys x))))
```

Somebody set us up the bomb!

```
 (defun talcum-tabdance-init ()
```

Be extra paranoid about the key. If auto-escalation is off and there is no escalation key, completion is effectively only selecting from the history.

```
  (if (not talcum-tabdance-key)
     (message "Tabdance not started: %s"
              "you have not defined an escalation key.")
    (define-key minibuffer-local-completion-map
     talcum-tabdance-key (quote talcum-tabdance-scalate))

    (when (and (talcum-latex-p)
               (not talcum-tabdance-no-hijacking))
     (~~tabdance-hijack))
    t))

 (defun ~~tabdance-hijack ()
  ;; Vaguely similar to the function from classic LaTeX mode as
  ;; of GNU Emacs 21.3. No "Options" Question -- who uses that
  ;; anyway? Display of default choice is standard now, offers
  ;; history, accommodates both latex-mode 21.3 and latex-mode
  ;; 22.
  (define-skeleton tex-latex-block
   "Create a matching pair of lines \\begin{NAME} and \\end{NAME} at point.
```

```
Puts point on a blank line between them. Offers history."
    (let ((blockname
         (completing-read
          (format "LaTeX environment: (default %s) " latex-block-default)
          (mapcar 'list (~~sparsed
                     (append (or
                            (~~value-safe standard-latex-block-names) ;; --21
                            (~~value-safe latex-standard-block-names));; 22--
                           latex-block-names)))
          (not ':filter-by-predicate)
          (not ':require-match)
          (not ':filled-out-already)
          'latex-block-names ; history
          latex-block-default ; default
          )))
      (setq latex-block-default blockname)
      blockname)
    ;; Don't start a new line for environment?
    ;; \n
    ;; Without extra >, we misindent for document.
    "\\begin{" str "}" >
    \n _ \n
    "\\end{" str "}" > \n)
```

In latex-mode 22–, C-c C-o points to latex-insert-block.

```
  (defalias 'latex-insert-block 'tex-latex-block))


 (defun talcum-tabdance (str pred allp)
  (let* ((realchoice (~~nth-filtered (1- tabdance-level)
                           tabdance-choices
                           'identity)))
    (cond

     ((equal t allp)
      (while (and talcum-tabdance-auto-escalate
                  (null (all-completions str realchoice pred))
                  (talcum-tabdance-escalate)
                  (setq realchoice (~~nth-filtered (1- tabdance-level)
                                       tabdance-choices
                                       'identity))))
      (all-completions str realchoice pred))

     ((null allp)
      (while (and talcum-tabdance-auto-escalate
                  (null (try-completion str realchoice pred))
                  (talcum-tabdance-escalate)
                  (setq realchoice (~~nth-filtered (1- tabdance-level)
                                       tabdance-choices
                                       'identity))))
      (try-completion str realchoice pred))

     ((equal 'lambda allp)
```

```
     (not (null (member str
                        (all-completions str realchoice pred)))))
    )))

(defun ~~nth-filtered (n xs pred)
 (nth n (~~filter pred xs)))

(defun ~~pseudolength (xs)
 (length (~~filter 'identity xs)))

(defadvice completing-read
 (around tabdance (prompt table
                   &optional predicate reqmatch initial hist def inherit)
       activate)
 (let ((tabdance-choices '())
      (table table)
      (tabdance-level 1)    ; Ugh. Think of something better.
      )

   (cond
    ;; Safeguard: we're not correctly set up, go back to old behaviour.
    ((or (not talcum-use-tabdance-flag)
         (not talcum-tabdance-key))
     t)

    ;; Explicit :tabdance request. Note that in this case, we are not
    ;; trying to be clever about the history. This is not needed,
    ;; because the caller already knows about tabdance, so it's
    ;; his/her responsibility to include history vars at a suitable
    ;; place.
    ((and (consp table)
          (equal (car table) ':tabdance))
     (setq tabdance-choices (cdr table)
           table 'talcum-tabdance))

    ;; Prefer the history.
    ((and talcum-tabdance-auto-prefer-history hist)
     (setq tabdance-choices (list
                             ;; We must convert the history from list
                             ;; to alist.
                             (mapcar 'list
                                     ;; The history may be a (symbol .
                                     ;; offset) cons as well, but we
                                     ;; always want the symbol's value,
                                     ;; which is a list.
                                     (symbol-value
                                      (if (consp hist)
                                          (car hist)
                                        hist)))
                             table)
           table 'talcum-tabdance))

    ;; Default case: do nothing fancy.
```

```
    (t
     t))
   ad-do-it))

(add-hook 'talcum-mode-hook (lambda () (if talcum-use-tabdance-flag
                                           (talcum-tabdance-init))))
)                                ; namespace
(provide 'talcum-tabdance)
;; talcum-tabdance.el ends here.
```

5 Ain't gotta compile them all!

5.1 talcum-pikeman.el

$Id: talcum-pikeman.el,v 1.16 2005/08/26 18:36:57 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-tools)
(require 'talcum-compat)

(talcum-namespace
 (~~sourcefile-versionstamp "$RCSfile: talcum-pikeman.el,v $" "$Revision: 1.16 $"))

(defgroup talcum-pikeman nil "The compiling options menu" :group 'talcum)

(talcum-namespace

 (defvar talcum-pikeman-compile-options-functions nil
   "A list of functions that will be called prior to calling tex-file.
Each function will be called with one argument, the name of the main
file, and should return a string. All these strings are concatenated,
seperated by spaces, and passed as parameters to TeX.")

 (defvar talcum-pikeman-compile-filestart-functions nil
   "A list of functions that will be called prior to calling tex-file.
Each function will be called with one argument, the name of the main
file, and should return a string. All these strings are concatenated,
and passed to TeX as the first input line, suitably quoted.")

 (defvar talcum-pikeman-menu-creator-functions nil
   "A list of functions that are called with no parameters to create
the Pikeman menu. They should return a list of menu items, since all
the results are appended.")

 (defun talcum-pikeman-generate-menu ()
  (list "--"
       (~~pikeman-generate-menu)))
```

```
(add-to-list 'talcum-menu-functions #'talcum-pikeman-generate-menu)

(defun ~~pikeman-generate-menu ()
 (list "Pikeman"
      :filter (lambda (_)
              (apply 'append
                   (mapcar
                    (lambda (mcf) (funcall mcf))
                    talcum-pikeman-menu-creator-functions)))))

(defun ~~pikeman-compiler-options (texfile)
  "Assemble the command-line options for the tex call (by calling
talcum-pikeman-compile-options-functions). TEXFILE is the name of the
main file, with extension and possibly path."
  (let ( (options  ; we do a little dance to exclude nil entries
       (mapconcat #'identity ; just to splice in the spaces
                (~~filter #'identity ; to remove the nils
                        (mapcar (lambda (optfun) (funcall optfun texfile))
                              talcum-pikeman-compile-options-functions))
                " ")))
    options))

(defun ~~pikeman-first-line (texfile)
  "Assemble the first line for the tex call (by calling
talcum-pikeman-compile-filestart-functions). TEXFILE is the name of
the main file, with extension and possibly path."
  (let ((firstline (mapconcat (lambda (filifun) (funcall filifun texfile))
                        talcum-pikeman-compile-filestart-functions "")))
    firstline))

(defun ~~pikeman-prepare-compiler-string ()
  "Set tex-command to include compiler options and the first line."
  (let* ((texfile (tex-main-file))
       (options (~~pikeman-compiler-options texfile))
       (firstline (~~pikeman-first-line texfile)))
    (setq tex-command
       (concat latex-run-command " "
             ;;"--jobname * " ;; * not quoted, replaced by tex-file
             (~~value-safe tex-start-options-string) ;; removed in 21--
             (~~value-safe tex-start-options) ;; 22+
             " "
             options " "
             " '"
             ;; somewhat ugly bugfix: what do we do about *
             ;; generated by tex-start-commands, tpc-filestart
             ;; et al.? If allowed, the \input below will fail,
             ;; since at most one * expansion is done.
             ;; * in options can still give replacement.
             ;; We assume existence of \def\@gobble#1{} (i.e. LaTeX 2e)
             "\\csname @gobble\\endcsname{*}"
             firstline
             (~~value-safe tex-start-commands) ;; 22+
             "\\input{"
```

```
                texfile
                "}'"))
    ))

;; Only needed for classic LaTeX mode, but who cares?
(defadvice tex-file (before talcum-pikeman activate)
 (~~pikeman-prepare-compiler-string))

;; Only for AucTeX. Can't check for AucTeX version since that is
;; sometimes a date, sometimes not.
(when (boundp 'TeX-expand-list)
 (add-to-list 'TeX-expand-list
     (list "%(Talcum-options)"
          (lambda () (~~pikeman-compiler-options (TeX-active-master t)))))
 (add-to-list 'TeX-expand-list
     (list "%(Talcum-line1)"
          (lambda () (~~pikeman-first-line (TeX-active-master t))))))

); namespace
```

That should be sufficient for the framework.

Now, the canonical examples (to be out-sourced à la TREX)

Ex. 1: Includes.

```
(talcum-namespace

;; to be made buffer-local? we want one per main file...
(defvar ~~pikeman-selected-chapters nil)
(defvar ~~pikeman-all-chapters nil)


(defun ~~pikeman-include-menu ()

 (setq ~~pikeman-all-chapters
      (~~pikeman-find-includes
       (or (and (boundp 'tex-main-file)
                tex-main-file)
           (and (talcum-auctex-p)
                (TeX-active-master t))
           nil)))
 (if ~~pikeman-all-chapters
     ;; if nothing to select, don't offer menu.
     ;; Caveat: we look at version on disk.
   (identity ;; was list for submenu, but that's a lot of mousing.
    (append '("--" "--"
              ["Included chapters:" nil]
              )
            (mapcar (quote ~~pikeman-menu-item) ~~pikeman-all-chapters)
            '("---"
              ["Include none"
               (talcum-namespace
                (setq ~~pikeman-selected-chapters nil))]
```

```
815
816             ["Include all"
817              (talcum-namespace
818               (setq ~~pikeman-selected-chapters ~~pikeman-all-chapters))]
819
820             ["Rescan file"
821              (talcum-namespace
822               (setq ~~pikeman-menu (~~pikeman-make-menu)))])
823           ))))
824
825  (defun ~~pikeman-menu-item (chapter)
826   `[,chapter
827     (talcum-namespace
828      (setq ~~pikeman-selected-chapters
829           (if (member ,chapter ~~pikeman-selected-chapters)
830              (remove ,chapter ~~pikeman-selected-chapters)
831            (cons ,chapter ~~pikeman-selected-chapters))))
832     :key-sequence nil          ;sorry, how would that work?
833     :style toggle
834     :selected (member ,chapter (quote ,(talcum-namespace ~~pikeman-selected-chapters)))
835     ])
836
837  (defun ~~pikeman-find-includes (&optional buffer-or-file)
838   (interactive "bMain buffer")
839   (let ((buffer (or (and (bufferp buffer-or-file) buffer-or-file)
840                 (and (stringp buffer-or-file)
841                     (let* ((bu (generate-new-buffer "Talcum temp master")))
842                       (save-excursion
843                         (set-buffer bu)
844                         (insert-file-contents buffer-or-file nil nil nil t)
845                         bu)))
846                 (current-buffer)))
847         (rst nil)
848         (inhibit-point-motion-hooks t)
849         (inhibit-redisplay t))
850     (save-excursion
851       (set-buffer buffer)
852       (goto-char (point-min))
853       (while (search-forward-regexp
854             "^[[:space:]]*\\\\ include{\\([^}]+\\)}"
855              (point-max) t)
856         (push (match-string-no-properties 1) rst))
857       (setq rst (nreverse rst))
858       ;;(message "%s" rst)
859       (if (stringp buffer-or-file)
860          (kill-buffer buffer))
861       rst)))
862
863  (defun ~~pikeman-make-includeonly (file)
864   ;; if we use all chapters, give no includeonly at all.
865   (unless (= (length ~~pikeman-all-chapters)
866              (length ~~pikeman-selected-chapters))
867     (concat "\\includeonly{"
```

```
          (mapconcat 'identity
                  ~~pikeman-selected-chapters
                  ",")
          "}")))


 (add-to-list 'talcum-pikeman-compile-filestart-functions
           (quote ~~pikeman-make-includeonly))
 (add-to-list 'talcum-pikeman-menu-creator-functions
           (quote ~~pikeman-include-menu))

 )                                ; namespace

;; Ex. 2: setting the interaction mode
(talcum-namespace

 (defvar ~~pikeman-interaction-mode nil)

 (defun ~~pikeman-interaction-menu ()
  (list (cons "Interaction mode"
       (~~menu-choose-one ~~pikeman-interaction-mode
                     '(("(default)" nil)
                      ("batch" "batchmode")
                      ("non stop" "nonstopmode")
                      ("scroll" "scrollmode")
                      ("errorstop" "errorstopmode"))))))

 (defun ~~pikeman-interaction-option (file)
  (if ~~pikeman-interaction-mode
     (concat "--interaction " ~~pikeman-interaction-mode)))

 (add-to-list 'talcum-pikeman-compile-options-functions
           (quote ~~pikeman-interaction-option))
 (add-to-list 'talcum-pikeman-menu-creator-functions
           (quote ~~pikeman-interaction-menu))
 ); namespace

;; Ex. 3: setting the format
(talcum-namespace

 (defcustom talcum-pikeman-formats
  '(("(default)" nil)
    ("pdfLaTeX" "pdflatex")
    ("LaTeX" "latex")
    ("Beamer" "beamer")
    ("Diplom" "diplom"))
  "Formats to offer in the Output Format menu. First element is the
menu name, second is the fmt file name."
  :group 'talcum-pikeman
  :type '(repeat
        (list (string :tag "Menu name")
             (choice :tag "Format"
                   (string :tag "File name")
```

```
                    (const :tag "(none)" nil)))))

 (defvar ~~pikeman-format nil)

 (defun ~~pikeman-format-menu ()
  (list (cons "Output format"
  ;(cons "---"
       (~~menu-choose-one ~~pikeman-format
                     talcum-pikeman-formats))))

(defun ~~pikeman-format-option (file)
 (if ~~pikeman-format
    (concat "&" ~~pikeman-format)))

 (add-to-list 'talcum-pikeman-compile-options-functions
           (quote ~~pikeman-format-option))
 (add-to-list 'talcum-pikeman-menu-creator-functions
           (quote ~~pikeman-format-menu))
);namespace

;; Ex. 4: \ listfiles .
(talcum-namespace
 (defvar ~~pikeman-listfiles nil)
 (defun ~~pikeman-listfiles-menu ()
  (list ;"--"
       '["\\ listfiles "
        (talcum-namespace (~~negate ~~pikeman-listfiles))
        :style toggle
        :selected (talcum-namespace ~~pikeman-listfiles)]))
 (defun ~~pikeman-make-listfiles (file)
  (if ~~pikeman-listfiles "\\listfiles"))

 (add-to-list 'talcum-pikeman-menu-creator-functions
           (quote ~~pikeman-listfiles-menu))
 (add-to-list 'talcum-pikeman-compile-filestart-functions
           (quote ~~pikeman-make-listfiles))
); namespace

;; Ex. 5: draft or final
(talcum-namespace
 (defvar ~~pikeman-draftmode nil)
 (defun ~~pikeman-draftmode-menu ()
  (list (cons "Draft/Final"
           (~~menu-choose-one ~~pikeman-draftmode
                         '(("(default)" nil)
                          ("draft" "draft")
                          ("final" "final"))))))
 (defun ~~pikeman-make-draftmode (file)
  (when ~~pikeman-draftmode
   (save-excursion
    (let ((bu (generate-new-buffer "Talcum temp master")))
     (set-buffer bu)
     (insert-file-contents file nil nil nil t)
```

```
        (goto-char (point-min))
        (if (search-forward-regexp "\\\\documentclass\\(\\[[^]]*\\]\\)?{\\([^}]+\\)}" nil t)
            ;; XXX: this fails for {...[...]...} within class options. Does anybody do that?
            (format "\\PassOptionsToClass{%s}{%s}"
                  ~~pikeman-draftmode
                  (or (match-string 2) ;; in case of options
                      (match_string 1)) ;; in case of no options
                  ))))

 (add-to-list 'talcum-pikeman-menu-creator-functions
           (quote ~~pikeman-draftmode-menu))
 (add-to-list 'talcum-pikeman-compile-filestart-functions
           (quote ~~pikeman-make-draftmode))
);namespace

;; Ex. 6: force non-embedding of base fonts into pdf.
(talcum-namespace
 (defvar ~~pikeman-nonembed-fonts nil)
 (defun ~~pikeman-nonembed-menu ()
  (list :"--"
        '["Don't embed base 14 fonts (PDF only)"
          (talcum-namespace (~~negate ~~pikeman-nonembed-fonts))
          :style toggle
          :selected (talcum-namespace ~~pikeman-nonembed-fonts)]))
 (defun ~~pikeman-make-nonembed (file)
  ;; Yeah, I know, this is not going to work for anybody else.
  (if ~~pikeman-nonembed-fonts "\\pdfmapfile{pdftex_ndl14}"))

 (add-to-list 'talcum-pikeman-menu-creator-functions
           (quote ~~pikeman-nonembed-menu))
 (add-to-list 'talcum-pikeman-compile-filestart-functions
           (quote ~~pikeman-make-nonembed))
 );namespace

;; Ex. 7: source specials.
(talcum-namespace
 (defvar ~~pikeman-srcspc nil)
 (defun ~~pikeman-srcspc-menu ()
  (list :"--"
        '["Source specials"
          (talcum-namespace (~~negate ~~pikeman-srcspc))
          :style toggle
          :selected (talcum-namespace ~~pikeman-srcspc)]))
(defun ~~pikeman-srcspc-option (file)
 (if ~~pikeman-srcspc
    "-src-specials"))

 (add-to-list 'talcum-pikeman-compile-options-functions
           (quote ~~pikeman-srcspc-option))
 (add-to-list 'talcum-pikeman-menu-creator-functions
           (quote ~~pikeman-srcspc-menu))

); namespace
```

```
(provide 'talcum-pikeman)
```

talcum-pikeman.el ends here.

6 Où sont les dvi d'antan?

6.1 talcum-prod.el

$Id: talcum-prod.el,v 2.5 2005/08/21 19:48:55 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-tools)
(require 'talcum-compat)
(require 'talcum-menu)
(require 'advice)

(defgroup talcum-prod nil "Prodding your viewer" :group 'talcum)

(defcustom talcum-use-prod-flag t
  "*Enable the whole prodding subsystem."
  :group 'talcum
  :type 'boolean)

;; current syntax is:
;; (predicate command)
(defcustom talcum-prod-commands
  '(((lambda () (string-match "xdvi" tex-dvi-view-command))
     "pkill -USR1 xdvi.bin")
    (ignore
     lower-frame))
  "A list of things that might be executed after LaTeXing.
Each element is a list (DOTHIS-P DOWHAT), where:
DOTHIS-P is t, nil, or a function. nil means do not execute this,
anything else means execute this.
DOWHAT is either a string or a function. If a function, it is called,
if a string, the string is fed into the shell that runs LaTeX."
  :group 'talcum-prod
  :type '(repeat
        (list
         (choice (const t) (function :value ignore))
         (choice (string :value "/bin/true") (function :value ignore))
         )))

(defcustom talcum-prod-cycle 1
  "Talcum checks if it can run the next prod command every n seconds."
  :group 'talcum-prod
  :type '(choice integer (number :value 0.3)))
```

```
(defvar talcum-prod-tick 0)

(talcum-namespace

 (defun talcum-prod-generate-menu ()
  (list "--"
       (~~menu-flag-toggle talcum-use-prod-flag
                           "Auto-refresh Viewer")))

 (add-to-list 'talcum-menu-functions #'talcum-prod-generate-menu)

 (defun ~~work-prod-queue (whatnext &optional docu)
  (when talcum-use-prod-flag
    ;; if delay is high, talcum might not be active anymore
    ;; when this triggers.
    (~~debug 0 "talcum-prod: advice entered, talcum mode is %s" talcum-mode)
    (when (or talcum-mode docu) ; evil hack: AucTeX moves to some other buffer
     (let ((dothis-p (nth 0 (car whatnext)))
           (dowhat (nth 1 (car whatnext)))
           (restjobs (cdr whatnext))
           (proc (cond
                ((talcum-latex-p) (tex-shell-proc))
                ((talcum-auctex-p) (TeX-process docu)))))
       (when (or
              (equal dothis-p t)  ; always
              (and (functionp dothis-p) (funcall dothis-p))) ; pred says go!
         (~~debug 0 "Yes, I'm doing something: %s" dowhat)
         ;; next, check if we have a shell string or a fun.
         (cond
          ((functionp dowhat)
           (~~debug 0 "Calling fun: %S" dowhat)
           (funcall dowhat))
          ((stringp dowhat)
           ;; big ugly hack: assume we can send command if
           ;; no new output is coming for 0.1s. Maybe this
           ;; will fail for files with lengthy calculations?
           ;; (fp comes to mind, as does beamer)
           ;; OTOH, on all but the first call, the other delay
           ;; is added to it as well.
           (if (or
                (and (talcum-latex-p)
                     (/= talcum-prod-tick
                         (setq talcum-prod-tick
                               (buffer-modified-tick (tex-shell-buf)))))
                ;;(accept-process-output proc 0 100))
                (and (or docu (talcum-auctex-p)) compilation-in-progress))
               (progn
                (~~debug 0 "no, not yet.")
                (setq restjobs whatnext))
             (~~debug 1 "Executing: %s" dowhat)
             ;; note: it's the user's responsibility to add "&"
             ;; if they want a background process here.
```

```
            (cond
             ((talcum-latex-p)
              (tex-send-command dowhat nil nil))
             ((or docu (talcum-auctex-p))
              (~~debug 0 "TeX-run-command %s %s %s -> %s" docu dowhat ""
                       (TeX-run-command docu dowhat "")))))
          (t (error "Unknown kind of dowhat: %s" dowhat))))
       (if restjobs
          (run-at-time talcum-prod-cycle nil
                       (quote ~~work-prod-queue) restjobs docu))))))

;; This is called upon activation of the mode.
(defun talcum-prod-init ()
  ;; This is easy: intercept tex-file.
  (cond
   ((talcum-latex-p)
    (defadvice tex-file (after talcum-prod-viewer) ; disable)
      "Prod viewers to refresh and show the new file."
      (~~work-prod-queue talcum-prod-commands))
    (ad-enable-advice 'tex-file 'after 'talcum-prod-viewer)
    (ad-activate 'tex-file)
    t)

   ;; This is tricky. Intercept the plethora of commands AucTeX provides.
   ((talcum-auctex-p)
    (defadvice TeX-run-format (after talcum-prod-viewer-AucTeX) ; disable)
      "Prod viewers to refresh and show the new file."
      (~~work-prod-queue talcum-prod-commands (ad-get-arg 0)))
    (ad-enable-advice 'TeX-run-format 'after 'talcum-prod-viewer-AucTeX)
    (ad-activate 'TeX-run-format)
    t)))

;; This is called upon deactivation of the mode.
(defun talcum-prod-exit ()
  (cond
   ((talcum-latex-p)
    (ad-disable-advice 'tex-file 'after 'talcum-prod-viewer)
    ;;(ad-activate 'tex-file)
    t)
   ((talcum-auctex-p)
    (ad-disable-advice 'TeX-run-format 'after 'talcum-prod-viewer-AucTeX)
    ;;(ad-activate 'TeX-run-format)
    t)))

(~~sourcefile-versionstamp "$RCSfile: talcum-prod.el,v $"
                           "$Revision: 2.5 $")

) ;;namespace

(add-hook 'talcum-enter-mode-hook #'talcum-prod-init)
(add-hook 'talcum-leave-mode-hook #'talcum-prod-exit)
```

```
(provide 'talcum-prod)
```

talcum-prod.el ends here

7 Rendering

7.1 talcum-render.el

$Id: talcum-render.el,v 2.20 2005/08/28 09:00:49 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-tools)
(require 'talcum-render-ui)
(require 'talcum-render-rules)

(talcum-namespace
 (~~sourcefile-versionstamp "$RCSfile: talcum-render.el,v $" "$Revision: 2.20 $"))

(defvar talcum-default-actions
  '(:property category
              talcum-rendered
    :property evaporate t
    :property priority 500 ;; default for show-paren-prio is 1000
;     :opener talcum-make-render-opener
     ))

 (defvar talcum-rex-init-hook nil
   "Hooks called at init time, before the rendering structures are
 initialized.")

(talcum-namespace
 (defvar ~~match-any-re)

 (defvar ~~rendermap (~~mmap-new))

 (defun talcum-render-string (str)
  "Add renderings to a string and return the result."
  (if (or (null str)
          (string= str ""))
      "`"
    (with-temp-buffer
      (with-syntax-table ~~mode-syntax-table
        (insert str)
        (talcum-render-region (point-min) (point-max) 'REPLACE)
        ;; We're actually carrying font-lock info all along, so don't
        ;; throw it away now :-D
        ;; (buffer-substring-no-properties (point-min) (point-max))
        (buffer-string)))))
```

```
;; Go through the region and call ~~maybe-apply-rule-at-point for
;; each match of any item according to the current ~~match-any-re.
;; ~~maybe-apply-rule-at-point must check itself whether the rule is
;; applicable. We leave point just after the match, i.e. after the
;; closing brace of a \begin{} or after the last char of a command.
;;
;; We save match data and excursion, so the rules are free to move
;; around and do stuff.
;;
;; The region to render is actually extended to the next following
;; non-word character (or end of buffer) for technical reasons.
(defun talcum-render-region (bor eor &optional replace?)
  (interactive "r")

  ;; Init thingy: if render-list is dirty, rebuild.
  (if (get (quote ~~rendermap) 'talcum-map-is-dirty)
      (talcum-render-reinit))

  (when (and talcum-use-render-flag
             (< bor eor))
    (~~save (excursion restriction)
      (~~without-touching-buffer

       (unless (~~value-safe ~~user-point)
         (setq ~~user-point (point)))

       (let* ((bor bor)
              ;; make sure we can see the character after a macro end, to
              ;; check if it's to be rendered.
              (eor (save-excursion
                     (goto-char eor)
                     (skip-syntax-forward "w")
                     (or (eobp)
                         (forward-char 1))
                     (point)))
              (final-point (copy-marker bor)) ;; always move forward

              new-end-or-nil
              matching-rules)

         (goto-char bor)

         (while (search-forward-regexp
                 ~~match-any-re
                 eor 'end-and-nil)

           (setq matching-rules
                 (~~mmap-get (concat (match-string 1)
                                     (match-string 2))
                             ~~rendermap)
                 new-end-or-nil nil)
```

```
          (move-marker final-point (or (match-end 1) (match-end 2)))

          (dolist (rule matching-rules)
            (unless new-end-or-nil
              (~~save (excursion match-data)
                      (setq new-end-or-nil
                            (~~maybe-apply-rule-at-point rule eor replace?))
                      (if new-end-or-nil
                          (move-marker final-point new-end-or-nil)))))
          (goto-char final-point))

        (setq ~~user-point nil)
        (set-marker final-point nil)
        t)))))

;; Check and apply RULE at this point.
;;
;; When called, we have point just after the matcher string,
;; match-data slot MATCH points at the match. We may move point and trash
;; match-data. By setting premature-abort to non-nil, we may prevent
;; other rules from looking at this as well.
;;
;; return what we think is a good point for rendering to continue.
(defun ~~maybe-apply-rule-at-point (rule scopelimit &optional replace?)
  (let* (args
         envlimit
         (pattern (~~pattern-part rule))
         (argspec (~~pattern-args pattern))
         (MATCH (if (equal ':macro (~~pattern-type pattern))
                    1
                  2))
         (startofpattern (match-beginning MATCH))
         (endofpattern (copy-marker (match-end MATCH))))

    (goto-char endofpattern)
    (if
        (and
         ;; This is cheap. Maybe some context-sensitive rule already prevents us.
         (~~rule-desired-p rule)

         ;; parse-arg. returns string if failure, eg. unclosed braces
         (not (stringp (setq args
                             (~~parse-arguments argspec scopelimit))))
         (set-marker endofpattern (point))

         ;; this would mess with the user's point
         (not (and (not replace?)
                   (~~value-safe ~~user-point)
                   (>= ~~user-point startofpattern)
                   (< ~~user-point endofpattern)))

         ;; if not environment, succeed, otherwise see if we can get the
         ;; end-of-environment, and then put it into the args.
```

```
1319          (if (not (equal ':environment (~~pattern-type pattern)))
1320              t
1321            (setq envlimit
1322                  (~~find-end-of-environment
1323                   (~~pattern-name pattern) scopelimit))
1324            (unless (stringp envlimit)
1325              ;;end of env in scope! (implicit nil otherwise)
1326              (set-marker endofpattern (point))
1327              ;;(message "considering `%sŽ" (buffer-substring startofpattern endofpattern))
1328              (~~append! args (list envlimit)))))  ;implicit t
1329
1330          ;; Yes! Go ahead and do it!
1331          ;; now, if somebody matched, always abort.
1332          (prog1 endofpattern
1333            (save-excursion
1334              (~~really-apply-rule-at-point rule args
1335                                            (cons startofpattern
1336                                                  (marker-position endofpattern))
1337                                            replace?)))
1338        ;; otherwise, hope for some other rule.
1339        nil)
1340      ))
1341
1342  ;; Really apply RULE at point.
1343  ;;
1344  ;; We know it's applicable and all the arguments are there.
1345  ;; PATTERNPOS is a cons with car just before and cdr just after the
1346  ;; whole thing we will hide. ARGS contains the arguments as strings,
1347  ;; and buffer positions for the environment body.
1348  (defun ~~really-apply-rule-at-point (rule args patternpos &optional replace?)
1349    (let* ((actions (append talcum-default-actions (~~action-part rule)))
1350           (localvars nil)
1351           (ov (make-overlay (car patternpos) (cdr patternpos)))
1352           (talcum-make-render-opener 'talcum-make-render-opener)
1353           action arg1 arg2 ;; current action and its args
1354           no-display-flag
1355           )
1356
1357      ;; run over the actions and do what they say.
1358      ;;
1359      (dolist (subov (overlays-in (car patternpos) (cdr patternpos)))
1360        (if (overlay-get subov 'display)
1361            (delete-overlay subov)))
1362
1363      (while actions
1364
1365        (setq action (pop actions))
1366        (cond
1367
1368         ((run-hook-with-args-until-success
1369           'talcum-apply-action-functions action 'actions ov))
1370
1371         ((eq action ':var) ;; Make a local var known.
```

```
 (setq arg1 (pop actions))
 (if (not (consp arg1))
    (~~var-scope-start arg1)
  (~~var-scope-start (car arg1))
  (set (car arg1) (cdr arg1))
  (setq arg1 (car arg1)))
 (push arg1 localvars))

((eq action ':eval) ;; eval the next element
 (eval (pop actions)))

((eq action ':opener)
 (setq talcum-make-render-opener (pop actions)))

((eq action ':face)
 (~~with-secretly-touching-buffer
  (add-text-properties
   (overlay-start ov) (overlay-end ov)
   (list 'face
        (cons (pop actions)
             (get-text-property (overlay-start ov)
                                'face))))))

((eq action ':property) ;; set the (overlay) property
 (overlay-put ov (pop actions) (pop actions)))

((eq action ':display) ;; set the display string
 (if replace?
    (~~with-secretly-touching-buffer
     (goto-char (car patternpos))
     (insert (~~eval-display-spec (pop actions) args))
     (delete-region (point) (min (+ (point)
                                    (cdr patternpos)
                                    (- (car patternpos)))
                                 (point-max))))
   (overlay-put ov 'display
            (format "%s" (~~eval-display-spec (pop actions)
                                              args)))))

((eq action ':recurse) ;; just render the body, without generating overlay
 (setq no-display-flag t)
 (let ((body (car (last args))))
   (talcum-render-region (car body) (cadr body))))

((eq action ':help) ;; set the flyover text
 (if replace? ;; doesn't make much sense if we only get the string.
    (pop actions)
   (overlay-put ov 'help-echo
            (format "%s" (~~eval-display-spec (pop actions)
                                              args)))))
(t (message "unknown action! %s" action))
))
```

```
    (if (string= (overlay-get ov 'display) "")
       (overlay-put ov 'display "<empty>"))

    (unless (or replace? no-display-flag)
      (funcall talcum-make-render-opener ov patternpos))
    (if no-display-flag
       (delete-overlay ov))

    ;; all done, remove local scope.
    (dolist (lv localvars)
      (~~var-scope-end lv))
    ))

;; turn a display spec into a string with text properties.
(defun ~~eval-display-spec (spec patternargs)
  (let* (result
       (render? (eq (car-safe spec) ':rendered))
       (spec (if render? (cadr spec)
             spec)))
    (setq result
        (cond
         ((run-hook-with-args-until-success
           'talcum-display-spec-functions spec patternargs))
         ((stringp spec) spec)
         ((integerp spec) (~~maybe-strip-parens (nth spec patternargs)))
         ((eq spec 'body)
          (let ((bodypos (car (last patternargs))))
            (buffer-substring (car bodypos)
                              (cadr bodypos))))
         ((functionp spec) (funcall spec patternargs))
         (t (error "Unknown result specifier %s" spec))
         ))
    (if render?
       (setq result (talcum-render-string result)))
    result))

;; The cheap, safe variant.
(defun talcum-make-render-opener (ov pos)
  (talcum-namespace
   (~~with-secretly-touching-buffer
    (let ((beg (overlay-start ov))
        (end (overlay-end ov)))
       (add-text-properties
        beg end
       `(point-entered (lambda (_ __)
                    (let* ((a (overlay-start ,ov))
                         (o (overlay-end ,ov)))
                      (when (and a o)
                        ;(message "a&o %s %s, point %s" a o (point))
                        (talcum-unrender-region a o)
                        ;(message "ok.")
                        t)
                      t)))
```

```
      )))))

(defun ~~render-open/close (ov enter-or-leave)
 (~~without-touching-buffer
  (~~with-secretly-touching-buffer
   (let* ((beg (overlay-start ov))
          (end (overlay-end ov)))
     (cond
      ((equal enter-or-leave 'enter)
       (overlay-put ov 'display nil)
       ;;      (overlay-put ov 'face nil)
       (font-lock-fontify-region beg end)
       (remove-text-properties
        beg end
        (list 'point-entered nil)))

      ((and (equal enter-or-leave 'leave)
            (null (overlay-get ov 'display))
            (overlay-start ov)
            (overlay-end ov)
            (eq (overlay-buffer ov) (current-buffer))
            (or (< (point) (overlay-start ov))
                (> (point) (overlay-end ov)))
            )
       (delete-overlay ov)
       (remove-text-properties
        beg end
        (list 'point-entered nil 'point-left nil))
       (talcum-render-region beg end)))))))



(defun ~~parse-arguments (argspec end)
  "Parse options ARGSPEC in current buffer. Each opening brace matches a
whole braced sexp, all but { are optional. * is optional. Any other
char matches itself. Return a list of what was parsed, or a string
detailing what went wrong. END limits how far in the buffer we will
go."
  (catch 'mismatch
    (let (result found)
      (dolist (wanted (append argspec nil)
                      (if (> (point) end)
                          (throw 'mismatch "Exceeded")
                        result))
        ;; we can always have whitespace between args.
        (skip-syntax-forward "-" end)
        (setq found (char-after))
        (if (null found) (throw 'mismatch "Premature EOB."))
        (~~append! result
                (list
                 (cond

                  ((run-hook-with-args-until-success
```

```
                    'talcum-parse-argument-functions wanted))

                  ((or (equal wanted ':mandatory)
                       (equal wanted ?{))
                   (with-syntax-table ~~sanitized-mode-syntax-table
                     (~~scantoken)))

                  ;; [ gets nothing or a [sexp]
                  ((or (equal wanted ':optional)
                       (equal wanted ?\[ ))
                   (if (= found ?\[)
                       (catch 'notclosed
                         (with-syntax-table ~~sanitized-mode-syntax-table
                           (let ((s (point))
                                 (forward-sexp-function nil))
                             (while (not (looking-at "[^{]*"))
                               (if (> (point) end)
                                   (throw 'notclosed ""))
                               (forward-sexp 1))
                             (search-forward "]")
                             (if (> (point) end)
                                 (throw 'notclosed ""))
                             (buffer-substring
                              s (point)))))
                     ""))

                  ;; special hardcode hack: * gets itself, optionally
                  ((or (equal wanted ':star)
                       (equal wanted ?*))
                   (if (= found ?*)
                       (buffer-substring
                        (point) (~~point-after (forward-char 1)))
                     ""))


                  ;; better not: more complaints = less bugs :D
                  ;; ;; any other char gets itself
                  ;; ((= wanted found)
                  ;;  (buffer-substring
                  ;;  (point) (~~point-after (forward-char 1))))

                  ;; if all else fails, complain and exit
                  (t (throw 'mismatch
                            (format "looking at %c and no rule matches."
                                    found)))))
                nil)))))

(defun ~~maybe-strip-parens (str)
  "Remove one level of enclosing parentheses from string, if any. We
also strip leading and trailing whitespace in that case. Note: we do
not check the parens match each other!"
  (if (and str (string-match "^[[:space:]]*\\s(\\([^\000]*\\)\\s)[[:space:]]*$" str))
      ;; evil hack there, but not handling NUL is better than not handling \n.
```

```
      (match-string 1 str)
    str))

 (defun ~~find-end-of-environment (envname end)
   "Find the end of the environment called envname.

We assume point after the \\begin, since somebody else might have
parsed arguments already. (Technically, point after the \\ is
 sufficient .) We return a list with the position of the first char in
and not in the env. body anymore. We leave point just after the
closing brace of the \\end We only check for nesting of this
environment, not any other. (This means we probably don't DTRT in the
preamble, if the begin-part of a newenvironment does a \\begin{foo}).
Returns a string if we run out of buffer before the env is closed."
   (catch 'endofbuffer
    (let ((depth 1)
          (re (concat "\\(\\\\begin\\|\\\\end\\){" envname "}"))
          (a (point))
           o)
      (while (and (not (zerop depth))
                  (or (search-forward-regexp re end t)
                      (throw 'endofbuffer "Exceeded scope")))
        ;(message "d%s, %s" depth (match-string 1))
        (if (string= (match-string 1) "\\begin")
            (setq depth (1+ depth))
          (setq depth (1- depth))))
      (list a
          (save-excursion
            (search-backward "\\end{")
            (point)))
      )))

 )                                    ;namespace

(provide 'talcum-render)
```

7.2 talcum-render-ui.el

$Id: talcum-render-ui.el,v 2.18 2005/08/28 09:00:49 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-tools)
(require 'talcum-render-rules)
(require 'talcum-custom)
(require 'talcum-menu)

(talcum-namespace
 (~~sourcefile-versionstamp "$RCSfile: talcum-render-ui.el,v $"  "$Revision: 2.18 $"))
```

```
1627
1628 (defcustom talcum-use-render-flag t
1629   "*Enable the whole rendering subsystem."
1630   :group 'talcum
1631   :type 'boolean)
1632
1633 (defvar talcum-render-list nil
1634   "The true render list .")
1635
1636 (talcum-namespace
1637  (defvar ~~mode-syntax-table nil)
1638  (defvar ~~sanitized-mode-syntax-table nil))
1639
1640 (defgroup talcum-render nil
1641   "Displaying beautiful letters instead of ugly macros"
1642   :group 'talcum)
1643
1644 (defcustom talcum-user-render-list nil
1645   "Local rendering rules"
1646   :group 'talcum-render
1647   :get 'talcum-render-rule-getter
1648   :set 'talcum-render-rule-setter
1649   :initialize 'custom-initialize-default
1650   :type talcum-render-rule-custom-type)
1651
1652 (defvar talcum-render-features 'nil)
1653
1654 (defvar talcum-apply-action-functions nil
1655   "Abnormal hook to introduce new action commands.
1656
1657 Each function is called with the overlay, the current action and the
1658  list of rest-args-and-further-actions. The function should not alter
1659 these unless it has verified it is applicable. The function should
1660 return a non-nil value only if no other rules shall apply to the
1661 action.")
1662
1663 (defvar talcum-display-spec-functions nil
1664   "Abnormal hook to introduce new display specifiers.
1665
1666 Each function is called with two parameters: the spec and a list
1667 containing the macro's arguments. The function returns a string with
1668 text properties as appropriate, or nil if it is not the right one.
1669 Note that you cannot prevent the :rendered syntax from working with
1670 this -- :rendered is hidden from you beforehand. You may, as a dirty
1671 hack, set the value of render? to nil to override :rendered, though,
1672 and to non-nil to simulate :rendered.")
1673
1674 (defvar talcum-parse-argument-functions nil
1675   "Abnormal hook used to introduce new kinds of macro parameter style.
1676
1677 Each function is called with one argument: the character that
1678  identifies the parameter style. Functions must return a non-nil value
1679 and leave point at the end of the parameter only if they have handled
```

```
1680 that parameter. Otherwise, they may not move point. If you have found
1681 your parameter to be there, but empty (like missing *, return the
1682 empty string \"\". ")
1683
1684 (defvar talcum-init-specpdl 6000
1685   "If you get \"exceeds max-specpdl-size\" errors on startup, set this
1686 to a higher value. (As comparison: GNU Emacs' default is 600.)")
1687
1688 (defvar talcum-semantic-braces
1689   '(?( ?) ?[ ?] ?« ?» ?< ?> ?\")
1690   "Not all braces should be made to match each other. For example, a)
1691 does not need an opening brace.")
1692
1693 (defgroup trexes
1694   nil                                   ;members
1695   "Customization options for Talcum Rendering EXtensions."
1696   :group 'talcum)
1697
1698 (defcustom talcum-render-trexes
1699   '((trex-math :math)
1700     (trex-unicode :unicode)
1701     (trex-markup :markup)
1702     (trex-xref :xref)
1703     (trex-sectioning :sectioning))
1704   "Trexes that will be loaded."
1705   :group 'trexes
1706   :type 'sexp)
1707
1708 (talcum-namespace
1709  ;; This used to be in talcum-render. Unfortunately, if font-lock
1710  ;; kicks in before talcum-render is completely loaded, we would get a
1711  ;; byte-compile warning: assignment to free var. I wouldn't give a
1712  ;; flying shit about that if it didn't switch-window away an actual
1713  ;; useful buffer with actual information in it. Thank you Emacs!
1714  (defvar ~~user-point nil)
1715  (make-variable-buffer-local (quote ~~user-point)))
1716
1717 ;; XXX HACK XXX should use the proper functions.
1718 ;;
1719 ;; 2005AUG21: Nevermind, from what I can see, all other font-lock
1720 ;; methods disappear in Emacs 22.
1721 (talcum-namespace
1722  (defadvice jit-lock-fontify-now
1723    (before talcum-user-point-setter activate)
1724    "Save user's point."
1725
1726    (unless (~~value-safe ~~user-point)
1727      (setq ~~user-point (point)))))
1728
1729
1730 (talcum-namespace
1731
1732  ;; initialize internal stuff.
```

```
1733  ;;
1734  ;; currently, this mainly recreates the internal hash structures and
1735  ;; regexpen. This should be called when entering the mode and
1736  ;; whenever talcum-render-list or talcum-render-features are changed.
1737  (defun talcum-render-reinit ()
1738    (interactive)
1739    (let* ((max-specpdl-size (max max-specpdl-size
1740                                  talcum-init-specpdl))
1741           matcher macro-matchers other-matchers)
1742
1743      (setq ~~rendermap (~~mmap-new))
1744      (with-temp-message "Building rendering structures..."
1745        (dolist (rule (append talcum-render-list
1746                              talcum-user-render-list))
1747          (when (~~rule-desired-p rule)
1748            (setq matcher (~~pattern-matcher (~~pattern-part rule)))
1749            (if (equal ':macro (~~pattern-type (~~pattern-part rule)))
1750                (push matcher macro-matchers)
1751              (push matcher other-matchers))
1752            (~~mmap-put matcher rule ~~rendermap)))
1753
1754        (setq ~~match-any-re
1755              (format "\\(%s\\)\\>\\|\\(%s\\)"
1756                      (if macro-matchers
1757                          (regexp-opt macro-matchers)
1758                        "\\000") ;; XXX hack, because regexp-opt is broken.
1759                      (if other-matchers
1760                          (regexp-opt other-matchers)
1761                        "\\000")))
1762        t)
1763      (message "Building rendering structures...done")
1764      (put (quote ~~rendermap) 'talcum-map-is-dirty nil)
1765      t))
1766
1767  (defun talcum-make-render-map-dirty ()
1768    (put (quote ~~rendermap) 'talcum-map-is-dirty t))
1769
1770  (defun talcum-render-init ()
1771
1772    (dolist (trex talcum-render-trexes)
1773      (require (car trex))
1774      (dolist (feat (cdr trex))
1775        (talcum-add-render-feature feat)))
1776
1777    (message "rex-hook")
1778    (run-hooks 'talcum-rex-init-hook)
1779    (font-lock-add-keywords nil '(talcum-render-fontlock-function) 'atend)
1780    (talcum-render-reinit)
1781    (setq ~~mode-syntax-table (copy-syntax-table))
1782    (setq ~~sanitized-mode-syntax-table (copy-syntax-table ~~mode-syntax-table))
1783    ;; it is stupid of forward-sexp et al. to look at ( and ) which are not
1784    ;; usually syntactical parens: \ texttt {)} would take the {) as its argument.
1785    (dolist (ch talcum-semantic-braces)
```

```
1786        ;; How the fuck does putting a R-val into a function have
1787        ;; side effects on the var with that R-val?
1788        (modify-syntax-entry ch "." ~~sanitized-mode-syntax-table))
1789      t)
1790
1791    (defun talcum-render-exit ()
1792      ;; no funs use timers anymore.
1793      (font-lock-remove-keywords nil '(talcum-render-fontlock-function))
1794      (talcum-unrender-region (point-min) (point-max))
1795      t)
1796
1797    (defun talcum-unrender-region (bor eor)
1798      "Remove the renderings created by talcum-render."
1799      (interactive "r")
1800      (save-excursion
1801        (~~without-touching-buffer
1802         (~~with-secretly-touching-buffer
1803          (remove-text-properties bor eor '(point-entered nil
1804                                            point-left nil
1805                                            face nil
1806                                            fontified nil )))
1807         (mapc (lambda (ov)
1808                 (if (equal (overlay-get ov 'category) 'talcum-rendered)
1809                     (delete-overlay ov)))
1810               (overlays-in bor eor)))))
1811
1812
1813    (defun talcum-render-find-suitable-start ()
1814      "Return a buffer position backwards from point where
1815    talcum-render-dwim should start."
1816      (~~without-touching-buffer
1817       (save-excursion
1818         (search-backward "%%%%%" (point-min) 'go-limit)
1819         (point))))
1820
1821    (defun talcum-render-find-suitable-end ()
1822      "Return a buffer position forward from point where
1823    talcum-render-dwim should start."
1824      (~~without-touching-buffer
1825       (save-excursion
1826         (search-forward "%%%%%" (point-max) 'go-limit)
1827         (point))))
1828
1829    (defun talcum-render-dwim (pfx)
1830      "Refresh the unclutterings in the current suitable region, as found
1831    by t-r-find-suitable-start, -end, which see. With prefix C-u, remove
1832    all unclutterings from that region."
1833      (interactive "P")
1834      (setq ~~user-point (point))
1835      (let ((bos (talcum-render-find-suitable-start))
1836            (eos (talcum-render-find-suitable-end)))
1837        (talcum-unrender-region bos eos)
1838        (unless (and (listp pfx) pfx)
```

```
        (talcum-render-region bos eos))))
 )                                          ;namespace


(defconst talcum-render-parsep-re "\\(\\\\par[^a-zA-Z]\\)\\|\\(\n\n\\)"
  "A regexp to match the beginning or end of a paragraph.")

(defun talcum-render-paragraph ()
  (interactive)
  "Render the paragraph point is in or after.
May you'll want to advice tex-terminate-paragraph to call this."
  (let* ((inhibit-point-motion-hooks t)
         (p (point)))
    (talcum-render-region
     (save-excursion (or (search-backward-regexp talcum-render-parsep-re (point-min) t)
                         (point-min)))
     (save-excursion (or (search-forward-regexp talcum-render-parsep-re (point-max) t)
                         (point-max))))))

(defun talcum-newpar-and-render ()
  (interactive)
  (let* ((inhibit-point-motion-hooks t))
    (save-excursion
      (talcum-render-paragraph))
    (insert "\n\n")))

(defun talcum-render-fontlock-function (limit)
  (talcum-render-region (point) limit)
  nil)

(defun talcum-render-fontf-function (start)
  (let ((end (save-excursion
               (catch 'end-of-buffer
                 (forward-char 600))
               (or (search-forward-regexp
                    talcum-render-parsep-re
                    (point-max)
                    'silent-go-limit)
                   (point-max)))))
    (message "rendering from %s to %s" start end)
    (talcum-render-region start end)
    t))
```

Provide keybindings

```
(define-key talcum-keymap "#" 'talcum-render-dwim)
(define-key talcum-keymap "\C-M" 'talcum-newpar-and-render)
```

Provide menu

```
(talcum-namespace
 (defun talcum-render-make-menu ()
  (list
   (~~menu-flag-toggle talcum-use-render-flag "Rendering"
                       (progn
```

```
                    (if talcum-use-render-flag
                       (talcum-render-init)
                     (talcum-render-exit))
                    (talcum-render-dwim 'on)))
    '["Render: DWIM" talcum-render-dwim]
    '["Render buffer" (talcum-render-region (point-min) (point-max))
     :included talcum-use-render-flag]
    '["Render region" (talcum-render-region (region-beginning) (region-end))
     :included talcum-use-render-flag]
    '["Unrender buffer" (talcum-unrender-region (point-min) (point-max))
     :included (not talcum-use-render-flag)]
    '["Unrender region" (talcum-unrender-region (region-beginning) (region-end))
     :included (not talcum-use-render-flag)]
    (let (trexmenu)
     ;; A submenu with a checkbox for every loaded TREX.
     (append '("TREXes" :active talcum-use-render-flag)
           (dolist (trex (talcum-render-features) (nreverse trexmenu))
             (push `[,(symbol-name trex)
                   (progn
                     (if (talcum-has-render-feature ,trex)
                        (talcum-remove-render-feature ,trex)
                      (talcum-add-render-feature ,trex))
                     (talcum-render-reinit))
                   :style toggle
                   :selected (talcum-has-render-feature ,trex)
                   ]
                 trexmenu))))
    ))
 (add-to-list 'talcum-menu-functions #'talcum-render-make-menu)
 )                              ; namespace

(add-hook 'talcum-enter-mode-hook #'talcum-render-init)
(add-hook 'talcum-leave-mode-hook #'talcum-render-exit)

(provide 'talcum-render-ui)
```

7.3 talcum-render-rules.el

$Id: talcum-render-rules.el,v 2.4 2005/03/06 22:22:43 ulmi RELEASE $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
;; '((type name args feature) action-part)

;; type is one of :macro :environment :active

;; :macros get a \ prepended, :env.s get a \begin{}, actives are left
;; unchanged

```

```
(require 'talcum-tools)

(talcum-namespace
 (~~sourcefile-versionstamp "$RCSfile: talcum-render-rules.el,v $" "$Revision: 2.4 $"))

(talcum-namespace
 (defconst ~~backslash "\\"
  "Regexp matching a backslash.")

 (defun ~~not-after-backslash (re)
  (concat "\\(?:^\\|[^" (regexp-quote ~~backslash) "]\\)" re))


 (defun ~~pattern-part (rule)
  (car rule))
 (defun ~~action-part (rule)
  (cdr rule))

 (defun ~~pattern-type (pattern)
  (nth 0 pattern))
 (defun ~~pattern-name (pattern)
  (nth 1 pattern))
 (defun ~~pattern-args (pattern)
  (nth 2 pattern))
 (defun ~~pattern-feat (pattern)
  (nth 3 pattern))

 (defun ~~pattern-matcher (pattern)
  "Make a matchable string from the pattern.

Note that we go through a level of regexp-quote before we search in
the buffer."
  (let ((name (~~pattern-name pattern))
        (type (~~pattern-type pattern)))
   (cond
    ((eq type ':macro)
     (concat ~~backslash name))
    ((eq type ':environment)
     (concat ~~backslash "begin{" name "}"))
    ((eq type ':active)
     name)
    (t
     (~~debug 9 "Huh? What's type %s of %s?" type name)
     nil))))

 ;; check a complex specification rule.
 ;; a sr can be conjunctive or disjunctive.
 ;; a csr is true iff all elements are fulfilled
 ;; a dsr is true iff not all elements are not fulfilled
 ;;
 ;; elements of a sr can be:
 ;; - integers : true iff not larger than talcum::thres
 ;; - functions: true iff they return not-nil
```

```
;; - lists      : lists within a dsr are csrs and vice versa
;; - keywords : true iff that keyword is in talcum::present-features
(defun ~~rule-desired-p (rule &optional disj)
  ;; we introduce boolean ops that guarantee to return t or nil to
  ;; prevent seeping out of information.
  ;; (also, we cannot do (funcall (if pred and or) a b) since and, or are sforms)
  (let ((&& (lambda (a b) (if a (if b t))))
        (|| (lambda (a b) (if a t (if b t)))))
    (~~rule-desired-p_ (~~pattern-feat (~~pattern-part rule)) disj)))

;; This function does the proper work for ~applicable. Splitting it up
;; prevents us from defining && and || over and over again.
;;
;; disj has a double role: it discriminates csr and dsr, and it must
;; also be the respective zero, i.e.
;; - nil for conjunction (since nil && x == nil for all x)
;; - t   for disjunction (since  t || x == t   for all x)
(defun ~~rule-desired-p_ (speclist &optional disj)
  (catch 'shortcut
    (let* ((applies? (not disj))
           (speclist (if (consp speclist) speclist (list speclist))))
      (dolist (feat speclist applies?)
        (setq applies?
              (funcall (if disj || &&)
                       (cond
                        ((eq t   feat) t)
                        ((eq nil feat) nil)
                        ((integerp feat) (>= talcum-render-threshold feat))
                        ((functionp feat) (funcall feat))
                        ((listp  feat) (~~rule-desired-p_ feat (not disj)))
                        ((keywordp feat) (talcum-has-render-feature feat))
                        (t            (error "unknown case of speclist: %s" feat)))
                       applies?))
        (if (eq applies? disj)
            (throw 'shortcut applies?))))))

;; XXX to be changed to use a hashtable.
(defun talcum-render-features ()
  (interactive)
  (message "%s" talcum-render-features)
  talcum-render-features)

(defun talcum-has-render-feature (feat)
  (memq feat talcum-render-features))

(defun talcum-add-render-feature (feat)
  (interactive "SFeature keyword:")
  (add-to-list 'talcum-render-features feat))

(defun talcum-remove-render-feature (feat)
  (interactive "SFeature keyword:")
  (setq talcum-render-features
        (delq feat talcum-render-features)))
```

```
); namespace

(provide 'talcum-render-rules)
```

8 It's all been done before.

8.1 trex-7bit.el

$Id: trex-7bit.el,v 1.2 2005/03/06 21:37:27 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-render)

(defvar trex-7bit-mappings
  '(("\"a" . "ä")
    ("\"o" . "ö")
    ("\"u" . "ü")
    ("\"s" . "ß")
    ("\"A" . "Ä")
    ("\"O" . "Ö")
    ("\"U" . "Ü")))

(dolist (ch trex-7bit-mappings)
  (setq talcum-render-list
        (cons `((:active ,(car ch) nil (:7bit :german))
                :display ,(cdr ch))
         talcum-render-list)))

(talcum-make-render-map-dirty)

(provide 'trex-7bit)
```

8.2 trex-lists.el

$Id: trex-lists.el,v 2.5 2005/03/06 21:37:55 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-render)

(defun trex-lists-arabic (i) (number-to-string i))
(defun trex-lists-alph (i) (char-to-string (+ 96 i)))
(defun trex-lists-Alph (i) (upcase (trex-lists-alph i)))

;; Let's hope nobody has lists with more than 39 roman items...
```

```
(defun trex-lists-roman (i)
 (let* ((result "")
        (i i))
   (while (>= i 10)
    (setq result (concat result "x")
          i (- i 10)))
   (concat result
         (nth i '("" "i" "ii" "iii" "iv" "v" "vi" "vii" "viii" "ix")))))

(defun trex-lists-Roman (i) (upcase (trex-lists-roman i)))

(setq talcum-render-list
    (append
     talcum-render-list
     '(((:environment "enumerate" "" (:lists))
        :var (trex-lists-nesting . (:enum 0))
        :recurse)
       ((:environment "itemize" "" (:lists))
        :var (trex-lists-nesting . (:item 0))
        :recurse)
       ((:environment "description" "" (:lists))
        :var (trex-lists-nesting . (:desc 0))
        :recurse)
       ((:macro "item" "[" (:lists))
        :display trex-lists-spitzmarke)
       )))
(talcum-make-render-map-dirty)


(defun trex-lists-spitzmarke (macparams)

 (if (not (boundp 'trex-lists-nesting))
    ;; Whoopsie! We're not in scope. Fallback.
    "** "
  (if (not (string= "" (nth 0 macparams)))
      ;; Explicitly given by user, use that
      (talcum-namespace (~~maybe-strip-parens (nth 0 macparams)))

    ;; all else fails: think for ourselves.
    (let* ((list-type (car-safe trex-lists-nesting))
           (nest-depth 0))

      (setq nest-depth
           (dolist (n (talcum-var-history 'trex-lists-nesting) nest-depth)
             (if (equal list-type (car-safe n))
                 (setq nest-depth (1+ nest-depth))))
           trex-lists-nesting (list (car trex-lists-nesting)
                                    (1+ (cadr trex-lists-nesting))))

      (cond
       ((equal :enum list-type)
        (funcall (nth nest-depth trex-lists-enum-style)
              (cadr trex-lists-nesting)))
```

```
        ((equal :item list-type)
         (funcall (nth nest-depth trex-lists-item-style)))

        ((equal :desc list-type)
         ;; An item of a description has an arg.
         "[nodesc?]")

        (t
         "??")))))


(defgroup trex-lists nil
  "Rendering itemize, enumerate and description."
  :group 'trexes)

(defcustom trex-lists-enum-style
  (list 'ignore ;; dummy element
    (lambda (i) (concat (trex-lists-arabic i) ". "))
    (lambda (i) (concat (trex-lists-alph i) ") "))
    (lambda (i) (concat (trex-lists-roman i) ". "))
    (lambda (i) (concat (trex-lists-Alph i) ". "))
    )
  "Functions that yield a list bullet with a number, one for each
nesting depth."
  :group 'trex-lists
  :type '(list
        (const ignore)
        function
        function
        function
        function
        ))

(defcustom trex-lists-item-style
  '(ignore ;; dummy element
    (lambda () "o ")
    (lambda () "- ")
    (lambda () "* ")
    (lambda () ". ")
    )
  "Functions that yield a list bullet without a number, one for each
nesting depth."
  :group 'trex-lists
  :type '(list
        (const ignore)
        function
        function
        function
        function
        ))
```

```
(provide 'trex-lists)
```

8.3 trex-markup.el

$Id: trex-markup.el,v 1.4 2005/03/06 21:38:14 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-tools)
(require 'talcum-render)

(defgroup trex-markup
  nil
  "Basic markup"
  :group 'trexes)

(defcustom trex-markup-macros
  '("textbf" "textit" "textsc"
         "textrm" "textsf" "texttt"
         "emph")
  "Basic markup macros. These take one argument, and that will be
displayed."
  :group 'trex-markup
  :type '(repeat string))

(dolist (rule trex-markup-macros)
  (add-to-list 'talcum-render-list
               `((:macro ,rule "{" (:markup))
                 :display (:rendered 0))))
(talcum-make-render-map-dirty)

(provide 'trex-markup)
```

8.4 trex-math.el

$Id: trex-math.el,v 2.6 2005/03/06 21:38:29 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-render)

(defgroup trex-math nil
  "Maths-specific rendering"
  :group 'trexes)

(defface trex-math-operator-face '((t (:weight bold :foreground "dark slate blue")))
  "The face math operators (like sin, cos, etc.) will be displayed in."
```

```
  :group 'trex-math)

(setq trex-math-operator-face 'trex-math-operator-face)

(defcustom trex-math-operators
  (list
   ;; "log-like symbols" from compr. LaTeX symbol list 08OCT2002
   "arccos" "arcsin" "arctan" "arg" "cos" "cosh" "cot" "coth"
   "csc" "deg" "det" "dim" "exp" "gcd" "hom" "inf"
   "ker" "lg" "lim" "liminf" "limsup" "ln" "log" "max"
   "min" "Pr" "sec" "sin" "sinh" "sup" "tan" "tanh")
  "\"log-like\" math operators"
  :group 'trex-math
  :type '(repeat string))

(defcustom trex-math-render-rules
  '(((:macro "big" "{" (:math))
     :property display (:height 1.2)
     :display 0)
    ((:macro "bigg" "{" (:math))
     :property display (:height 1.3)
     :display 0)
    ((:macro "Big" "{" (:math))
     :property display (:height 1.45)
     :display 0)
    ((:macro "Bigg" "{" (:math))
     :property display (:height 1.6)
     :display 0)

    ((:macro "left" "{" (:math))
     :property display (:height 1.3)
     :display 0)
    ((:macro "right" "{" (:math))
     :property display (:height 1.3)
     :display 0)
    ((:macro "middle" "{" (:math))
     :property display (:height 1.3)
     :display 0)

    ((:active "\\{" nil (:math))
     :display "{")
    ((:active "\\}" nil (:math))
     :display "}")
    )
  "Maths-related rules"
  :group 'trex-math
  :get 'talcum-render-rule-getter
  :set 'talcum-render-rule-setter
  :initialize 'custom-initialize-default
  :type talcum-render-rule-custom-type)

(setq talcum-render-list
      (append talcum-render-list
```

```
            trex-math-render-rules
            (mapcar
             (lambda (op)
               `((:macro ,op nil (:math))
                 :face trex-math-operator-face :display ,op))
             trex-math-operators)
            ))
(talcum-make-render-map-dirty)

(provide 'trex-math)
```

8.5 trex-sectioning.el

$Id: trex-sectioning.el,v 1.2 2005/03/06 22:23:10 ulmi RELEASE $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-render)

(defvar trex-sectioning-section-commands
  '("part" "chapter" "section"
    "subsection" "subsubsection"
    "paragraph" "subparagraph"
    ))

(dolist (sec trex-sectioning-section-commands)
  (setq talcum-render-list
        (cons `((:macro ,sec "*[{" :sectioning)
              :display 2)
         talcum-render-list)))

(talcum-make-render-map-dirty)

(provide 'trex-sectioning)
```

8.6 trex-unicode.el

$Id: trex-unicode.el,v 2.7 2005/03/06 21:38:45 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-tools)
(require 'talcum-render)

(defgroup trex-unicode nil
  "Rendering macros as unicode characters"
  :group 'trexes)
```

```
(defface trex-unicode-face '((t (:slant normal)))
  "A face known to contain many unicode glyphs.

This face is applied to characters by :unicode rules to prevent them
from getting lost in italics, boldface, etc."
  :group 'trex-unicode)

(setq trex-unicode-face 'trex-unicode-face)

(defcustom trex-unicode-mappings
  '( ;; format is (macro . utf16code)
    ;; Unicode Math Ops 2200–22FF
    ;; Much still missing.
    ("forall"      . #x2200)
    ("complement"  . #x2201)
    ("partial"     . #x2202)
    ("exists"      . #x2203)
    ("emptyset"    . #x2205)
    ("nabla"       . #x2207)
    ("in"          . #x2208)
    ("notin"       . #x2209)
    ("ni"          . #x220b)
    ("qedhere"     . #x220e)
    ("prod"        . #x220f)
    ("coprod"      . #x2210)
    ("sum"         . #x2211)
    ("mp"          . #x2213)
    ("setminus"    . #x2216)
    ("circ"        . #x2218)
    ("cdot"        . #x2219)
    ("sqrt"        . #x221a)
    ("infty"       . #x221e)
    ("land"        . #x2227)
    ("wedge"       . #x2227)
    ("lor"         . #x2228)
    ("vee"         . #x2228)
    ("cap"         . #x2229)
    ("cup"         . #x222a)
    ("int"         . #x222b)
    ("iint"        . #x222c)
    ("iiiint"      . #x222d)
    ("neq"         . #x2260)
    ("ne"          . #x2260)
    ("leq"         . #x2264)
    ("le"          . #x2264)
    ("geq"         . #x2265)
    ("ge"          . #x2265)
    ("prec"        . #x227a)
    ("succ"        . #x227b)
    ("subset"      . #x2282)
    ("supset"      . #x2283)
    ("subseteq"    . #x2286)
```

```
("supseteq"    . #x2287)
("subsetneq"   . #x228a)
("supsetneq"   . #x228b)
("unlhd"       . #x22b4)
("lhd"         . #x22b2)
("unrhd"       . #x22b5)
("rhd"         . #x22b3)
;; From U27F0-U27FF Supplemental Arrows A
("implies"     . #x27f9)
("iff"         . #x27fa)
("mapsto"      . #x27fc)
;; to is uncertain: function-goes-to is long, limit-to short
("to"              . #x27f6)
("longleftarrow"  . #x27f5)
("longrightarrow"     . #x27f6)
("longleftrightarrow"  . #x27f7)
("Longleftarrow"      . #x27f8)
("Longrightarrow"     . #x27f9)
;; From U2190-U21FF Arrows
("leftarrow"          . #x2190)
("uparrow"            . #x2191)
("rightarrow"         . #x2192)
("downarrow"          . #x2193)
("leftrightarrow"     . #x2194)
("updownarrow"        . #x2195)
;; many omitted...
;; From U2000ff General Punctuation
("dots"               . #x2026)
("ldots"              . #x2026)
("textperthousand"    . #x2030)
;; From 27D0ff Misc Math Symb A
;; currently all omitted
;; From U2A00ff Supp Math Op
("bigodot"            . #x2a00)
("bigoplus"           . #x2a01)
("bigotimes"          . #x2a02)
("lneq"               . #x2a87)
("gneq"               . #x2a88)
;;
("wp"  . #x2118)
("ell"  . #x2113)
("Im"  . #x2111)
("Re"  . #x211c)
("Finv"  . #x2132)
("Game"  . #x2141)
("aleph"  . #x2135)
("beth"  . #x2136)
("gimel"  . #x2137)
("daleth"  . #x2138)
;; Greekery. Note it is not technically correct to use
;; the low chars, but who has fonts that have the 1D400
;; Math Alnums? (Italic bold sans-serif fraktur
;; lowercase greek letter chi my ass!)
```

```
    ("alpha" . #x03b1)
    ("beta" . #x03b2)
    ("gamma" . #x03b3)
    ("delta" . #x03b4)
    ("epsilon" . #x03f5)
    ("varepsilon" . #x03b5)
    ("zeta" . #x03b6)
    ("eta" . #x03b7)
    ("theta" . #x03b8)
    ("vartheta" . #x03d1)
    ("iota" . #x03b9)
    ("kappa" . #x03ba)
    ("varkappa" . #x03f0)
    ("lambda" . #x03bb)
    ("mu" . #x03bc)
    ("nu" . #x03bd)
    ("xi" . #x03be)
    ("pi" . #x03c0)
    ("varpi" . #x03d6)
    ("rho" . #x03c1)
    ("varrho" . #x03f1)
    ("sigma" . #x03c3)
    ("varsigma" . #x03c2)
    ("tau" . #x03c4)
    ("upsilon" . #x03c5)
    ("varphi" . #x03c6)
    ("phi" . #x03d5)
    ("chi" . #x03c7)
    ("psi" . #x03c8)
    ("omega" . #x03c9)
    ("digamma" . #x03dd) ;;XXX is \digamma lowercase?
    ("Gamma" . #x0393)
    ("Delta" . #x0394)
    ("Theta" . #x0398)
    ("Lambda" . #x039b)
    ("Xi" . #x039e)
    ("Pi" . #x03a0)
    ("Sigma" . #x03a3)
    ("Upsilon" . #x03a5)
    ("Phi" . #x03a6)
    ("Psi" . #x03a8)
    ("Omega" . #x03a9)
    )
  "An alist of macro-to-utf16-code mappings."
  :group 'trex-unicode
  :type '(repeat
          (cons
           (string)
           (integer)))
  )

(defun trex-unicode-make-sample ()
  (interactive)
```

```
  (dolist (c trex-unicode-mappings)
    (insert "\\" (car c)
            " ~= " (decode-char 'ucs (cdr c))
            "\n"))
  (insert "\nand the number sets:\n")
  (dolist (c (list "C" "H" "N" "P" "Q" "R" "Z"))
    (insert c " ~= " (trex-unicode-mathbb (list c)) "\n"))
  t)


(defun trex-unicode-init ()
  ;;   (font-lock-add-keywords
  ;;    nil
  ;;     '(("\\\\[ a-zA-Z@]+" 0 trex-unicode-face prepend))
  ;;    'append)
  t)
(add-hook 'talcum-rex-init-hook 'trex-unicode-init 'atend)

(defun trex-unicode-display-function (spec args)
  (when (eq (car-safe spec) ':unicode)
    (propertize
     (char-to-string (decode-char 'ucs
                                  (cadr spec)))
     ':face 'trex-unicode-face
     'fontified t)))

(add-hook 'talcum-display-spec-functions
          'trex-unicode-display-function)

(dolist (c trex-unicode-mappings)
  (setq talcum-render-list
        (cons `((:macro ,(car c) nil (:unicode))
              :face trex-unicode-face
              :display (:unicode ,(cdr c)))
              talcum-render-list)))

(setq talcum-render-list
      (cons
       '((:macro "mathbb" "{" (:unicode))
         :display trex-unicode-mathbb)
       talcum-render-list))

(talcum-make-render-map-dirty)


(defun trex-unicode-mathbb (args)
  (talcum-namespace
   (or
    (~~switch (~~maybe-strip-parens (car-safe args))
              ("C" (char-to-string (decode-char 'ucs #x2102)))
              ("H" (char-to-string (decode-char 'ucs #x210d)))
              ("N" (char-to-string (decode-char 'ucs #x2115)))
              ("P" (char-to-string (decode-char 'ucs #x2119)))
```

```
            ("Q" (char-to-string (decode-char 'ucs #x211a)))
            ("R" (char-to-string (decode-char 'ucs #x211d)))
            ("Z" (char-to-string (decode-char 'ucs #x2124))))
    (car-safe args)
    "?")))

(provide 'trex-unicode)
```

8.7 trex-verbatim.el

$Id: trex-verbatim.el,v 1.2 2005/03/06 21:39:00 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-render)

(defun trex-verbatim-parse-arg-any-repeated-delimiter (wanted)
  (when (equal wanted ':any-repeated)
    (let ((delim-char (char-after))
          (b (point)))
      (forward-char 1)
      (if (search-forward (char-to-string delim-char) (point-max) t)
          (buffer-substring b (point))
        (goto-char b)))))

(add-hook 'talcum-parse-argument-functions
          'trex-verbatim-parse-arg-any-repeated-delimiter)

(defvar trex-verbatim-render-rules
  '(((:macro "verb" (:star :any-repeated) :verbatim)
     :face font-lock-constant-face
     :display (lambda (args) (substring (cadr args) 1 -1)))
    ((:environment "verbatim" nil :verbatim)
    :face font-lock-constant-face
    :display body)
    ))


(setq talcum-render-list
      (append trex-verbatim-render-rules
              talcum-render-list))
(talcum-make-render-map-dirty)


(provide 'trex-verbatim)
```

8.8 trex-xref.el

$Id: trex-xref.el,v 1.5 2005/03/06 21:39:38 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-tools)
(require 'talcum-render)
(require 'talcum-custom)

(defgroup trex-xref nil
  "Cross-references: index, label, cite, ref and footnote."
  :group 'trexes)

(defcustom trex-xref-rules
  '(((:macro "label" "{" (:xref))
     :display "^"
     :help 0)
    ((:macro "cite" "[{" (:xref))
     :display "[cit]"
     :help 1)
    ((:macro "ref" "{" (:xref))
     :display "/^"
     :help 0)
    ((:macro "eqref" "{" (:xref))
     :display "(/^)"
     :help 0)
    ((:macro "index" "{" (:xref))
     :display "[ix]"
     :help 0)
    ((:macro "footnote" "{" (:xref))
     :display "Ž"
     :help 0))
  "Rules defined by this trex."
  :group 'trex-xref
  :type talcum-render-rule-custom-type
  :get 'talcum-render-rule-getter
  :set 'talcum-render-rule-setter
  :initialize 'custom-initialize-default)

(setq talcum-render-list (append talcum-render-list
                         trex-xref-rules))

(talcum-make-render-map-dirty)

(provide 'trex-xref)
```

9 Mixed implementation cruft

9.1 talcum-compat.el

$Id: talcum-compat.el,v 2.6 2005/02/28 17:23:37 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
2586
2587 (require 'talcum-tools)
2588
2589 (talcum-namespace
2590  (~~sourcefile-versionstamp "$RCSfile: talcum-compat.el,v $" "$Revision: 2.6 $"))
2591
2592
2593 (defun talcum-infidel-p ()
2594 "Returns t for XEmacs, nil otherwise."
2595 (string-match "XEmacs\\|Lucid" (emacs-version)))
2596
2597 (defun talcum-auctex-p ()
2598 "Returns t if AucTeX is running, nil otherwise"
2599 (and (boundp 'AUC-TeX-version) (eq major-mode 'latex-mode)))
2600
2601 (defun talcum-latex-p ()
2602 "Returns t if plain old latex mode is running, nil otherwise"
2603 (and (not (boundp 'AUC-TeX-version)) (eq major-mode 'latex-mode)))
2604
2605 (defcustom talcum-buffer-molestors
2606   '( reftex-view-crossref-when-idle
2607      reftex-view-cr-ref
2608      reftex-view-cr-cite
2609      latex-indent
2610      ;;show-paren-function ;; won't highlight rendered paren otherwise
2611      )
2612   "A list of function symbols of functions that do not set
2613 inhibit-point-motion-hooks even though they should.
2614
2615 This can lead to unwanted unrendering."
2616   :type '(repeat function))
2617
2618 (dolist (fun talcum-buffer-molestors)
2619   (message "disabling in: %s %s" fun (symbolp fun))
2620   (eval
2621    `(defadvice ,fun (around talcum-point-motion-hook-disabler activate)
2622     "Prevent point-motion-hooks in this function."
2623     (let ((inhibit-point-motion-hooks t))
2624       ad-do-it))))
2625
2626 (provide 'talcum-compat)
2627
2628 ;; talcum-compat.el ends here
```

9.2 talcum-tools.el

$Id: talcum-tools.el,v 2.7 2005/08/12 16:11:31 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(defmacro talcum-namespace_ (nspc &rest what)
  ;;    "Make NSPC the namespace prefix in WHAT. You can use ~~ in
  ;; symbols and it will be replaced by NSPC. For example,
  ;; (talcum-namespace \"mess\" (~~age \"foo!\") becomes (progn
  ;; (message \"foo!\"))."
  (let (rst name expr (what what))

    (while (equal (car-safe (car-safe what)) 'macro)
      (setq what (macroexpand what)))

    (while what
      (setq expr (pop what))
      (cond

       ;; A symbol. Maybe we need to substitute.
       ((symbolp expr)
        (push (if (and (< 1 (length (setq name (symbol-name expr))))
                       (string= "~~" (substring name 0 2)))
                  (intern (concat nspc (substring name 2)))
                expr)
              rst))

       ;; A non-empty list. We have excluded the case of a macro by
       ;; the loop at the top. Do the car and cdr recursively and put
       ;; them together again. Be careful to strip the added progn and
       ;; list nesting by cadr. (This took a while to get right.)
       ((and (consp expr) (not (listp (cdr expr))))
        (push
         (let* ((x (cadr
                    (macroexpand `(talcum-namespace_ ,nspc ,(car expr)))))
                (y (cadr
                    (macroexpand `(talcum-namespace_ ,nspc ,(cdr expr))))))
           (cons x y))
         rst))

       ((consp expr)
        (push (cdr (macroexpand `(talcum-namespace_ ,nspc ,@expr))) rst))

;;       ((consp expr)
;;        (push
;;         (cons
;;          (cadr (macroexpand `(talcum-namespace_ ,nspc ,(car expr))))
;;          (cadr (macroexpand `(talcum-namespace_ ,nspc ,(cdr expr)))))
;;          rst))

       ;; It's a literal or nil or something like that.
       ;; Just push it over.
```

```
      ((push expr rst))))

   (cons 'progn (nreverse rst))))

;; The default namespace for internal Talcum use is "\ talcum\ \ ".
;; Identifier names with spaces in them should be pretty rare, so
;; we will probably not collide.
(defmacro talcum-namespace (&rest what)
  `(talcum-namespace_ " talcum " ,@what))

;; We might want for some reason to disable the namespacing.
(talcum-namespace
 (defmacro ~~no-namespace (&rest what)
   `(talcum-namespace_ "~~" ,@what)))


(defcustom talcum-verbosity 5
  "*How much going-on Talcum should send to *Messages*.
0 means everything (good for debugging), 10 is silence of the lambdas."
  :group 'talcum
  :type '(integer))


(talcum-namespace
```

write one save, get save-excursion, -match-data, etc. We do not actually check any semantics, or the presence of the corresponding save-foos.

(save (foo bar baz) bang) expands to (save-foo (save-bar (save-baz bang)))

```
 (defmacro ~~save (things &rest cmds)
   (let (result
         (things (nreverse things)))
     (setq result (cons (intern (concat "save-"
                                        (symbol-name (pop things))))
                        cmds))
     (dolist (thing things result)
       (setq result (list (intern (concat "save-"
                                          (symbol-name thing)))
                          result)))))



 (defmacro ~~append! (l &rest ls)
   "Append all args and setq the first to the result."
   `(setq ,l (append ,l ,@ls)))

 (defmacro ~~point-after (fu)
   "Execute FU and return the (possibly new) value of point."
   `(progn ,fu (point)))
);namespace
(talcum-namespace
 (defmacro ~~without-touching-buffer (&rest cmds)
   "Do something without touching the buffer, i.e. it is read-only.
Just because we're extra paranoid and this is probably not very expensive."
```

```
  `(let ((buffer-read-only t)
         (inhibit-read-only nil)
         (inhibit-redisplay t)
         (inhibit-point-motion-hooks t)
         (inhibit-modification-hooks t))
     ,@cmds))

 (defmacro ~~with-secretly-touching-buffer (&rest cmds)
   "Modify the current buffer, but do not change read-only or
buffer-modified flags. This is for the rare situation where you want
to override without-touching-buffer."
   `(let ((inhibit-read-only t)
         (deactivate-mark nil)
         (was-virgin (buffer-modified-p)))
     (prog1 (progn ,@cmds) (set-buffer-modified-p was-virgin))))
 );namespace
(talcum-namespace

 (defun ~~scantoken (&optional kind)
   "Move point forward by a token, and return the token. Whitespace
after the token is skipped, just as (La)TeX does. Optional parameter
KIND can be 'cmd or 'env; if 'env, we actually skip a whole \\begin{foo}."
   (let ((bot (point))
        (eot nil)
        (forward-sexp-function nil) ;;latex-mode thinks it's clever. It's not.
        )

     (skip-chars-forward " \t")
     (if (eq kind 'env)
        (~~scantoken 'cmd))        ; get the "\begin"
     (setq eot
          (cond
           ((looking-at "\\\\")     ; a control sequence proper
            (forward-char 1)
            (if (not (looking-at "[a-zA-Z]")) ;; self-terminating: \$ etc.
               (~~point-after (forward-char 1))
              (skip-chars-forward "a-zA-Z") ;; proper command. skip trailing blanks.
              (prog1
                 (point)
                (skip-chars-forward " \t"))))
           ((looking-at "{")
            (condition-case nil
               (~~point-after
                (forward-sexp))
              (scan-error (setq eot (1+ bot)))
              )) ;FIXME
           (t
            (~~point-after (forward-char 1)))))

     (buffer-substring bot eot)))

 (defun ~~fwd-arg ()
   "Go forward until a semi-properly nested exp has been moved over.
```

```
This means there may be spurious non-matching closing parens."
  (catch 'fail
    (let (stack)
      (skip-syntax-forward "^(")
      (push (char-after) stack)
      (forward-char 1)
      (while stack
        (skip-syntax-forward "^()")
        (if (looking-at "\\s(")
            (push (char-after) stack)
          (if (equal (char-after)
                     (cdr (aref (syntax-table) (car stack))))
              (pop stack)
            (if (eobp) (throw 'fail nil))))
        (forward-char 1))
      t)))
);namespace
```

Multimaps

```
(talcum-namespace
 ;; create and return a new multimap.
 (defun ~~mmap-new ()
  (make-hash-table :test 'equal))

 ;; put a key-value entry into a multimap.
 ;; This does not remove current entries in the multimap,
 ;; even if they have the same key.
 (defun ~~mmap-put (key val mmap)
  (puthash key (cons val (gethash key mmap nil)) mmap))

 ;; look up all entries for a key in a multimap.
 ;; returns nil if none present.
 (defun ~~mmap-get (key mmap)
  (gethash key mmap nil))

 (defun ~~mmap-keys (mmap)
  (let (result)
    (maphash (lambda (key val) (push key result)) mmap)
    result))
); namespace
```

Scoped Variables.

```
(talcum-namespace
(defun ~~var-scope-start (name)
 (let ((stack (get name 'talcum-varstack)))
  (if (boundp name)
      (push (symbol-value name) stack))
    ;(make-local-variable name)) nope, render-string opens tempbuffer
  (put name 'talcum-varstack stack)
  (set name nil)))
```

```
(defun ~~var-scope-end (name)
 (let ((stack (get name 'talcum-varstack)))
  (put name 'talcum-varstack (cdr-safe stack))
  (if (null stack)
     (makunbound name)
    (set name (car-safe stack)))))

(defun talcum-var-history (name)
 (let ((stack (get name 'talcum-varstack)))
  (and (boundp name)
      (cons (symbol-value name) stack))))

);namespace
```

Generic tools.

```
(talcum-namespace
 (defun ~~filter (p xs)
  (let ((r nil))
   (if (not (listp xs))
      (error "Trying to filter %S which is not a list" xs)
     (if (not (functionp p))
        (error "Trying to filter by %S which is not a function" p)
       (while xs
        (let ((x (car xs))
             (xss (cdr xs)))
          (setq r (if (funcall p x) (cons x r)
                 r))
          (setq xs xss)))
       (nreverse r)))))

 (defun ~~debug (severity &rest args)
  "Do debugging output, depending on talcum-verbosity."
  (if (>= severity talcum-verbosity)
     (apply 'message args)))


 (defun ~~sourcefile-versionstamp (id rev)
  (setq talcum--versions
      (cons (list id rev)
          (if (boundp 'talcum--versions)
             talcum--versions
            nil))))


 (defmacro ~~switch (expr &rest choices)
  "Cond template. Each element of CHOICES is compared against the value of EXPR in a cond."
  (let ((temp (make-symbol "--switchvar--")))
  `(let* ((,temp ,expr))
    ,(cons 'cond
         (mapcar
          (lambda (choice)
           (if (consp choice)
```

```
              `((equal ,temp ,(car choice))
                ,(cadr choice))
             `(t ,choice)))
          choices)))))

 (defmacro ~~value-safe (symbol)
  (if (boundp symbol) symbol
    nil))


 (~~sourcefile-versionstamp "$RCSfile: talcum-tools.el,v $"  "$Revision: 2.7 $")

); namespace

(provide 'talcum-tools)
```

talcum-tools.el end here

9.3 talcum-custom.el

$Id: talcum-custom.el,v 1.4 2005/02/19 18:13:59 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

```
(require 'talcum-tools)

(defconst talcum-render-rule-custom-type
  '(repeat                          ; many rules
    (list :tag "rule"                  ; one rule
          (list :tag "pattern part"     ; the pattern part

                (choice :tag "type"   ; the type part
                        :value :macro
                        (const :value :macro)
                        (const :value :environment)
                        (const :value :active))

                (string :tag "name"
                        :value "textbf") ; the name

                (choice :tag "argspec"   ; the argspec
                        :value "*[{"
                        (string :tag "short form"
                                :value "*[{") ; short argspec

                        (repeat :tag "long form"
                                (symbol :value :mandatory) ; long argspec
                                ))
                (choice :tag "featspec" ; featspec
```

```
;; FIXME: featspec needs extra attention to disambiguate
;; feature keywords and function results; likewise for
;; "never" rule and nested lists? Or drop nil rule altogether?
;; OTOH, empty lists aren't really meaningful here

(const :tag "always"
       :value t) ; – success
(const :tag "never"
       :value nil) ; – failure
(integer :tag "render–level"
         :value 3) ; – render–thres
(list :tag "feature"
      (const :value --feature)
      (symbol :tag "feature"
              :value ":trex–my–feature")
      )             ; – feature
(list :tag "ask predicate"
      (const :value --function)
      (function :tag "ask predicate"
                :value ignore)) ; – function
(repeat :tag "all of the following"  ; – recursive featspec
        (choice :tag "all of the following"
         (const :tag "always"
                :value t) ; –– success
         (const :tag "never" ; –– failure
                :value nil)
         (integer :tag "render–level"
                  :value 3) ; –– render–thres
         (list :tag "feature"
               (const :value --feature)
               (symbol :tag "feature"
                       :value ":trex–my–feature")
               ) ; –– feature
         (list :tag "ask predicate"
               (const :value --function)
               (function :tag "ask predicate"
                         :value ignore)) ; –– function
         (repeat :tag "any of the following" ; –– recursive featspec ii
                 (choice
                  (const :tag "always" t) ; ––– success
                  (const :tag "never" nil) ; ––– failure
                  (integer :tag "render–level"
                           :value 3) ; ––– render–thres
                  (list :tag "feature"
                        (const :value --feature)
                        (symbol :tag "feature"
                                :value ":trex–my–feature")
                        ) ; ––– feature
                  (list :tag "ask predicate"
                        (const :value --function)
                        (function :tag "ask predicate"
                                  :value ignore)) ; ––– function
                  (repeat :tag "deeper nesting (freeform)"
```

```
                                        (sexp))))
                        ))
                ))

        (repeat                     ;the action part
         :tag "action part"
         :value (:display "*")

         (choice
          :value (:display "*")

          (list :tag "display"
               :value (:display "*")

               (const :value :display)
               (choice            ; proper display flavour
                :value "*"

                (string :tag "fixed text"
                        :value "*")
                (integer :tag "(n+1)th argument"
                         :value 0)
                (const :tag "body of environment"
                       :value body)
                (list :tag "(rendering recursively)"
                     (const :tag ":rendered"
                            :value :rendered)
                     (choice      ; recursive display flavour
                      :value "*"
                      (string :tag "fixed text"
                              :value "*")
                      (integer :tag "(n+1)th argument"
                               :value 0)
                      (const :tag "body of environment"
                             :value body body)
                      (function :tag "result of function"
                                :value ignore)))
                (function :tag "result of function"
                          :value ignore)))

          (list :tag "evaluate anything"
               :value nil
               (const :value :eval)
               (sexp))

          (list :tag "force face"
               (const :value :face) ; face action
               (face))

          (list :tag "fly-over help"
               (const :value :help)
               (choice            ; display flavour
                (string :tag "fixed text"
```

```
                    :value "*")
               (integer :tag "(n+1)th argument"
                        :value 0)
               (const :tag "body of environment"
                      :value body body)
               (function)))

         (list :tag "render-opener"
               (const
                :value :opener)
               (function
                :value ignore))

         (list :tag "property"
               (const
                :value :property)
               (symbol :tag "name"
                       :value prop)
               (sexp :tag "value"
                     :value nil))

         (list :tag "local variable"
               :value my-local-var
               (const :value :var)
               (choice
                (variable :tag "name"
                          :value my-local-var)
                (cons :tag "with default value"
                      (variable :tag "name"
                                :value my-local-var)
                      (sexp :tag "value"
                            :value ""))))
         ))
        ))
  "A custom-compatible semi-representation of rules. Only works in
conjunction with the special getter talcum-render-rule-getter and
setter talcum-render-rule-setter, which see."
  )


;; Custom vars for rules should look like this:
;;  (defcustom foorules nil "" :get 'talcum-render-rule-getter :set 'talcum-render-rule-setter :type talcum-render-rule-custon

(defconst talcum-render-actions-arities
  '((:display . 1)
    (:eval   . 1)
    (:face   . 1)
    (:help   . 1)
    (:opener . 1)
    (:property . 2)
    (:var    . 1))
  "Alist that gives the number of arguments for each display flavour.

```

Not keeping this list up−to−date will prevent customize from working
for unlisted flavours.")

Circumvent limitations of customization interface.
Here be dragons! At least, -getter isn't one sexp with nested mapcar lambdas anymore.

```
(talcum-namespace

 ;; FIXME: this should set dirty flag?
 (defun talcum-render-rule-setter (nam val)
  "Customize−setter for render rule lists ."
  (set nam
       (mapcar (lambda (l) (cons (~~render-rule-pattern-setter (car l))
                                 (apply 'append
                                        (apply 'append (cdr l)))))
               val)))

 (defun talcum-render-rule-getter (nam)
  "Customize−getter for render rule lists ."
  (mapcar (quote ~~render-rule-getter1)
          (symbol-value nam)))

 (defun ~~render-rule-getter1 (rule)
  (let ((rule rule)
        pattern
        newactions)
    (setq pattern (~~render-rule-pattern-getter (pop rule))) ; pattern part
    ;; now treat the action part
    (while rule
     (let ((action nil)
           i)
       (push (pop rule) action) ;; action name
       (dotimes (i (cdr (assoc (car action)
                               talcum-render-actions-arities)))
         (push (pop rule) action))
       (push (nreverse action) newactions)))
    (list pattern (nreverse newactions))))

 (defun ~~render-rule-pattern-getter (pattern)
  (list (nth 0 pattern)
        (nth 1 pattern)
        (nth 2 pattern)
        (~~render-rule-pattern-getter1 (nth 3 pattern))))

 (defun ~~render-rule-pattern-getter1 (pattern)
  (cond
   ((functionp pattern) (list '--function pattern))
   ((listp pattern) (mapcar (quote ~~render-rule-pattern-getter1)
                            pattern))
   ((keywordp pattern) (list '--feature pattern))
   (t pattern)))

 (defun ~~render-rule-pattern-setter (pattern)
```

```
  (list (nth 0 pattern)
        (nth 1 pattern)
        (nth 2 pattern)
        (~~render-rule-pattern-setter1 (nth 3 pattern))))

 (defun ~~render-rule-pattern-setter1 (pattern)
  (cond
   ((listp pattern)
    (if (member (car pattern) '(--feature --function))
       (cadr pattern)
      (mapcar (quote ~~render-rule-pattern-setter1)
            pattern)))
   (t pattern)))
 )                                ; namespace

(talcum-namespace
 (~~sourcefile-versionstamp "$RCSfile: talcum-custom.el,v $" "$Revision: 1.4 $"))

(provide 'talcum-custom)
```

9.4 talcum-menu.el

$Id: talcum-menu.el,v 1.5 2005/08/26 18:37:41 ulmi Exp $

This file is part of Talcum.

Talcum is released under the terms and conditions of the LaTeX Project Public License 1.3a (see file LPPL-13a for details)

Talcum has the maintenance status: author-maintained. Maintainer: Ulrich M. Schwarz, ulmiusers.sarovar.org.

Finally: a clickable menu.

The entries are generally runtime-generated. This would not be necessary for all kinds of menu, but for some (think chapter selection), and I think it should not pose a big run-time overhead.

```

(require 'talcum-tools)
(require 'talcum-custom)

(talcum-namespace
 (~~sourcefile-versionstamp "$RCSfile: talcum-menu.el,v $" "$Revision: 1.5 $"))

(talcum-namespace

 (defmacro ~~negate (var)
  `(setq ,var (not ,var)))
```

Some common menu helpers

A checkbox for a variable.

```
 (defmacro ~~menu-flag-toggle (var str &optional more)
  `[,str
    (talcum-namespace (~~negate ,var) ,more)
    :style toggle
    :selected ,var
```

```
    ])
```

A radio field. Names come from the cars, values come from the cadrs.

```
 (defmacro ~~menu-choose-one (var names-and-vals)
   `(mapcar
    (lambda (nav)
      (vector (car nav)
            (list 'setq ',var (cadr nav))
            :style 'radio
            :selected (list 'equal ',var (cadr nav))))
    ,names-and-vals))

 (defvar talcum-menu-functions nil
   "A list of functions. Each function is called with no arguments.
The result should be a menu item definition. All results are
concatenated and used as the Talcum menu.")

 ;; Not sure if this will be needed. Menus are strange, unclear if
 ;; it's safe to change talcum-menu's value directly.
 (defun talcum-menu-regenerate ()
  (interactive)
  (easy-menu-define talcum-menu talcum-mode-map
    "Talcum's menu."
    (cons "Talcum"
          (apply 'append
           (mapcar 'funcall talcum-menu-functions))))
  t)

 (talcum-menu-regenerate)

) ; namespace
(provide 'talcum-menu)
```

Index