```
;; The beginning...
;;
;;
;;Global functions
;;
;;move x-from y-from w h x-to y-to
;;
;;allgemeine move-routine für text-blöcke
;;
;;
;;Global Objects
;;
;;Cursor sys-cursor
;;
;;
;;
;;
;;
;;
;;Objects
;;
;;Cursor
;;
;;int xpos ypos xpos-mem ypos-mem
;;
;;move-down max-y
;;move-up min-y
;;move-left min-x
;;move-right max-x
;;
;;
;;
;;Window
;;======
;;int x y w h
;;
;;(x y) Koordinaten obere linke Ecke
;;(w h) Width Height
;;
;;
;;
;;WindowWithCursor
;;================
;;Window
;;Cursor
;;
;;
;;TextWindow
;;==========
;;Window
;;Cursor
;;Text text
;;Point cursor-point
;;(Point) aux-point-list
;;
;;Text
;;====
;;(Line) lines
;;int num-lines
;;
;;
;;
```



```
;;Line
;;====
;;String text
;;int line-len
;;int left-start
;;
;;;; for managing marked parts of text
;;int mark-start
;;int mark-end
;;

;;Point
;;=====
;;pointer into (Line) of Text: line
;;int pos  ;; 0 <= pos <= (line-len line)
;;int pos-cnt
;;
;;move-up
;;move-down
;;
;;move-left
;;move-right
;;
;;insert-at char c
;;insert-at string str
;;
;;break-line-at
;;
;;copy
;;
;;
;;
;;Window  <-------> WindowClass
;;
;;Window ------> WindowClassFunction
;;    IndividualWindowData
;;
;;WindowClassFunction receives WindowHandle = Window as a lisp pointer as parame
ter
;;thereby gets access to individual Window Data
;;
;;Example
;;
;;TextWindow
;;
;;| WindowWithCursor | text | cursor-point | aux-point-list |
;;
;;
;;An Event is a structure of
;;
;;|event|ch|modif|
;;
;;integer event
;;ch and modif general lisp pointers
;;
;;for key events
;;ch and modif have the function explicated in *keyb01*
;;
;;event = 1: key press event
;;event = 2: key release event
;;
;;
```

```lisp
;;
;;Edit Window
;; |      |
;; |      |
;; |      |
;; |   Display Control
;; |      |
;; |   ___/
;; |  /
;; | |
;;Text
;;
;;

(defstruct (point-2d (:conc-name pt2d-))

(x 0)
(y 0)
)


;;
;; Textmanager
;;

(defconstant +no-marking+ 0)
(defconstant +marking+ 1)

(defstruct (textmanager (:conc-name tm-))

(text (make-text))
(mark-state +no-marking+)

(caret (freeze-pt (get-point)))
(mark-start (freeze-pt (get-point)))

(modif-start (freeze-pt (get-point)))
(modif-end (freeze-pt (get-point)))

(preferred-x 0)

move-code

)


;;
;; Window
;;

(defstruct (window (:conc-name win-))

(textm (make-textmanager))

(upper-left-pt (get-point))

(cursor (make-point-2d))
(cursor-old (make-point-2d))
)


;;
```



```lisp
;; Point
;;

(defstruct (point (:conc-name pt-))

line
pos-in-line

frozen

)


;;
;; Line
;;

(defstruct line

prev
next

buffer

len
start-visible

line-num

mark-start
mark-end
)


;;
;; Text
;;

(defstruct (text (:conc-name txt-))

first-line
last-line
num-lines)


;;
;; the move-codes
;;



(defconstant +up+ 0)
(defconstant +down+ 1)
(defconstant +left+ 2)
(defconstant +right+ 3)

(defconstant +pg-up+ 4)
(defconstant +pg-down+ 5)

(defconstant +home+ 6)
(defconstant +end+ 7)
```

```
;;
;; the init-functions initialize already created structures
;;

(defun init-text (text)
  (setf (txt-first-line text) nil)
  (setf (txt-last-line text) nil)
  (setf (txt-num-lines text) 0)
  text)

(defun init-text-line (text line)
  (setf (txt-first-line text) line)
  (setf (txt-last-line text) line)
  (setf (txt-num-lines text) 1)
  text)

(defun init-textmanager (textm)
  (setf (tm-text textm) (init-text (tm-text textm)))
  (setf (tm-mark-state textm) +no-marking+)
  (setf (tm-caret textm) (freeze-pt (get-point)))
  (setf (tm-mark-start textm) (freeze-pt (get-point)))
  textm)


(defun copy-to-buffer (buffer str)
  (do ((i 0 (+ i 1)))
      ((>= i (length str)))
    (setf (char buffer i) (char str i))))

(defun init-line-string (line str)
  (setf (line-prev line) nil (line-next line) nil)
  (setf (line-buffer line) (make-string 133))

  (copy-to-buffer (line-buffer line) str)

  (setf (line-len line) (length str))

  (setf (line-start-visible line) (compute-start-visible line))
  (setf (line-line-num line) 0)

  (setf (line-mark-start line) 0)
  (setf (line-mark-end line) 0)
  line)


(defun compute-start-visible (line)
  0)


;;
;; text functions
;;

(defun is-empty-text (text)
  (and (null (txt-first-line text)) (null (txt-last-line text))))


;; updates pos-in-text in lines following pt

(defun update-line-num (pt)
  (let ((akt-line (pt-line pt)))
```



```
    (let ((akt-line-num (+ (line-line-num akt-line) 1)))
      (do ((pline (line-next akt-line) (line-next pline)))
          ((null pline))
        (setf (line-line-num pline) akt-line-num)
        (setf akt-line-num (+ akt-line-num 1))))))

;; insert line after pt where pt points into a text

(defun insert-in-text (text pt line)
  (let ((pline (pt-line pt)))
    (let ((follow (line-next pline)))
      (setf (line-next pline) line)
      (setf (line-prev line) pline)
      (if follow
          (progn
            (setf (line-next line) follow)
            (setf (line-prev follow) line))
        (progn
          (setf (txt-last-line text) line)))
      (update-line-num pt))))


(setf *linecnt* 0)

(defun append-text-line (text line)
  (if (is-empty-text text)
      (init-text-line text line)
    (progn
      (insert-in-text text (line-to-pt (txt-last-line text)) line))))


;;
;;
;; textm functions
;;


(defun append-textm-line (textm line)
  (append-text-line (tm-text textm) line))

(defun append-textm-string (textm str)
  (append-textm-line textm (init-line-string (make-line) str)))


;;
;; the edit window callback
;;
;;


(defun callback-edit-window (win event ch modif)

  (let ((textm (win-textm win)))

    (cond ((is-ascii-char ch)

           (textm-insert-char textm ch))

          ((is-vk-down ch)

           (textm-down textm modif))
```

```
          ((is-vk-up ch)

           (textm-up textm modif))

          ((is-vk-left ch)

           (textm-left textm modif))

          ((is-vk-right ch)

           (textm-right textm modif))

          ((is-carriage-return ch)

           (textm-insert-cr textm modif))

          ((is-backspace ch)

           (textm-backspace textm))

          ((is-delete ch)

           (textm-delete textm))

          ((is-tab ch)

           (textm-insert-tab textm modif))

          ((is-home ch)

           (textm-home textm modif))

          ((is-end ch)

           (textm-end textm modif))

          ((is-pg-up ch)

           (textm-pg-up textm modif))

          ((is-pg-down ch)

           (textm-pg-down textm modif)))

    (update-display win textm)))


(defun pt-to-x-y (ul-pt akt-pt)
  (let ((line-ul (pt-line ul-pt))
        (line-ap (pt-line akt-pt)))
    (let ((ln-ul (line-line-num line-ul))
          (ln-ap (line-line-num line-ap)))
      (let ((y (- ln-ap ln-ul)))
        (let ((x (- (xpos-disp akt-pt) (xpos-disp ul-pt))))
          (values x y))))))


;;
;;
;; update upper left pt
;;
```



```
(defun update-upper-left-pt (ul-pt cursor move-code)
  )

;;
;; the delegating move functions
;;

(defun textm-right (textm modif)
  (textm-move textm modif +right+))


(defun textm-left (textm modif)
  (textm-move textm modif +left+))


(defun textm-up (textm modif)
  (textm-move textm modif +up+))


(defun textm-down (textm modif)
  (textm-move textm modif +down+))


(defun textm-pg-up (textm modif)
  (textm-move textm modif +pg-up+))


(defun textm-pg-down (textm modif)
  (textm-move textm modif +pg-down+))


(defun textm-home (textm modif)
  (textm-move textm modif +home+))


(defun textm-end (textm modif)
  (textm-move textm modif +end+))


;;
;; the non delegating functions, operating on textm
;;

(defun textm-insert-char (textm ch)
  )

(defun textm-insert-tab (textm modif)
  )


(defun textm-backspace (textm)
  )

(defun textm-delete (textm)
  )

;;
;; the point functions
;;
;;
;;
```

```
;;
;; freeze-pt makes point pt persistent in pool
;; thaw-pt makes pt reclaimable in pool
;; not implemented yet
;;



(defun freeze-pt (pt)
  pt)

(defun thaw-pt (pt)
  pt)

;;
;; get-point allocates points from a pool
;; functionality not implemented yet
;;

(defun get-point ()
  (make-point))

;;
;; (point-leq pt1 pt2) is true when pt1 <= pt2 in the text they point to
;;




(defun point-leq (pt1 pt2)
  (let ((line-num-1 (line-line-num (pt-line pt1)))
        (line-num-2 (line-line-num (pt-line pt2))))
    (or (< line-num-1 line-num-2)
        (and (= line-num-1 line-num-2) (<= (pt-pos-in-line pt1) (pt-pos-in-line
pt2))))))

(defun point-eq (pt1 pt2)
  (and (eq (pt-line pt1) (pt-line pt2)) (= (pt-pos-in-line pt1) (pt-pos-in-line
pt2))))


;; a point copy

(defun point-copy (pt)
  (let ((res (get-point)))
    (setf (pt-line res) (pt-line pt))
    (setf (pt-pos-in-line res) (pt-pos-in-line pt))
    res))



;;
;; return points pointing to line begin / line end of of line pointed to by line
 pt
;;


(defun line-anf (pt)
  (let ((res (point-copy pt)))
    (setf (pt-pos-in-line res) 0)
```



```
    res))


(defun line-end (pt)
  (let ((res (point-copy pt))
  (end-line-pos (- (line-len (pt-line pt)) 1)))
    (setf (pt-pos-in-line res) end-line-pos)
    res))


;;
;; is true wenn pt is in line pointed to by line-pt
;;

(defun in-line (pt line-pt)
  (eq (pt-line pt) (pt-line line-pt)))


;; converts line to pt pointing to line

(defun line-to-pt (line)
  (if (null line)
      nil
    (progn
      (let ((pt (get-point)))
        (setf (pt-line pt) line)
        (setf (pt-pos-in-line pt) 0)
       pt))))


;; moves pt one line up resp. one line down


(defun line-up (pt)
  (line-to-pt (line-prev (pt-line pt))))

(defun line-down (pt)
  (line-to-pt (line-next (pt-line pt))))


;; computes x displacement of pt in line pointed to by pt

(defun is-tab-char (ch)
  (= (char-code ch) 9))

(defun xpos-disp (pt)
  (let* ((line (pt-line pt))
         (buffer (line-buffer line)))
    (do ((i 0 (+ i 1))
         (xpos 0 xpos))
        ((>= i (pt-pos-in-line pt)) xpos)
      (setf xpos (next-xpos-ch xpos (char buffer i))))))

(defun next-xpos-ch (xpos ch)
  (if (is-tab-char ch)
      (align (+ xpos 1) 4)
    (+ xpos 1)))

(defun align (x n)
  (if (= (mod x n) 0)
      x
    (* (+ (floor x n) 1) n)))
```

```lisp
;;
;; returns moved point
;;


(defun move-point (pt move-code preferred-x)
  )


(defun move-caret (textm move-code)



;;
;; extends the marked area when caret moves from old-pt to new-pt
;;


(defun extend-mark (textm old-pt new-pt)

  ;; two flags describing directions

  (let ((move-caret-up (point-leq new-pt old-pt))
        (akt-line-pt (line-anf new-pt))
        (akt-pt (point-copy old-pt)))

    ;; go through full lines lying between old-pt and new-pt

    (freeze-pt akt-line-pt)

    (do ((next-akt-pt nil next-akt-pt))
        ((in-line akt-pt akt-line-pt))

      (if move-caret-up
          (setf next-akt-pt (line-anf akt-pt))
        (setf next-akt-pt (line-end akt-pt)))

      (reverse-mark move-caret-up akt-pt next-akt-pt)

      ;; move-caret-up = t informs reverse-mark that next-akt-pt <= akt-pt

      (if move-caret-up
          (setf akt-pt (move-point next-akt-pt +left+))
        (setf akt-pt (move-point next-akt-pt +right+))))

    (thaw-pt akt-line-pt)

    (reverse-mark move-caret-up akt-pt new-pt)))




;;
;;
;; the caret move only function, parameterized by move-code
;;


(defun textm-move (textm modif move-code)
```



```lisp
  (let* ((old-caret (tm-caret textm))
   (new-caret (move-caret textm move-code))

    (cond ((shift-pressed modif)

     (cond ((is-state textm +marking+)
       (extend-mark textm old-caret new-caret))
      (t
       (create-mark textm old-caret new-caret))))

    (t
     (cond ((is-state textm +marking+)
       (clear-mark textm)))))

    (thaw-pt old-caret)
    (freeze-pt new-caret)

    (setf (tm-move-code) move-code)
    (setf (tm-caret textm) new-caret)))




;;
;; the test output pane as a CAPI pane
;;
;;
;;

(defun terminal-input (self x y gspec)

  (let ((data (sys:gesture-spec-data gspec))

        (mods (sys:gesture-spec-modifiers gspec)))


    (gp:draw-string

     self

     (format nil "data = ~a mods = ~a~%" data mods)

;      (with-output-to-string (ss)

;        (sys:print-pretty-gesture-spec

;         gspec ss :force-shift-for-upcase nil))

     x y)))



(defclass terminal-pane (capi:output-pane)

  ()

  (:default-initargs

   :input-model '((:gesture-spec
```

```
                  terminal-input))

   :display-callback 'redraw-terminal))


(defparameter *max-lines* 54)
(defparameter *max-cols* 133)


(setf *terminal-buffer* (make-array (list *max-lines* *max-cols*) :initial-eleme
nt #\Space ))


(setf *test-win* (make-window))

(defun init-win (win fname)
  (setf (win-textm win) (init-textmanager (win-textm win)))
  (with-open-file (str fname :direction :input)
    (loop
     with line-str = nil
     do
     (setf line-str (read-line str nil nil nil))
     (when line-str
       (append-textm-string (win-textm win) line-str))
     until (null line-str))
    (setf (win-upper-left-pt win) (line-to-pt (txt-first-line (tm-text (win-text
m win)))))
    (freeze-pt (win-upper-left-pt win))))




(defun fill-buffer-from-pt (pt)
  (loop
   for i from 0 to (1- *max-lines*) do
   (fill-buffer-pt i pt)
   (setf pt (line-down pt))
   until (null pt)))

(defun fill-buffer-pt (i pt)
  (let ((line (pt-line pt)))
    (loop
     for j from 0 to (min (1- (line-len line)) (1- *max-cols*)) do
     (setf (aref *terminal-buffer* i j) (char (line-buffer line) j)))))




(defun redraw-terminal (pane x y width height)

  (loop for i from 0 to (1- *max-lines*) do
    (loop for j from 0 to (1- *max-cols*) do
      (gp:draw-character pane (aref *terminal-buffer* i j) (* j 12) (+ (* i 13)
13)))))


(defun showit ()
  (init-win *test-win* "lispm-editor.lisp")
  (fill-buffer-from-pt (line-to-pt (txt-first-line (tm-text (win-textm *test-win
*)))))
  (capi:contain (make-instance 'terminal-pane :title "Terminal")))
```



```
(defun displayit ()
  (loop
   with pline = (txt-first-line (tm-text (win-textm *test-win*)))
   do
   (format t "~a ~a ~a~%" (line-line-num pline) (line-len pline) (subseq (line-bu
ffer pline) 0 (line-len pline)))
    (setf pline (line-next pline))
   until (null pline)))



(defun ntabs (n)
  (let ((s (make-string n)))
    (loop for i from 0 to (1- n) do (setf (char s i) #\Tab))
    s))

(setf *str* (concatenate 'string (ntabs 3) "wuerfel und kugel" (ntabs 2) "kreis und kegel
"))

(setf *testline* (make-line))

(init-line-string *testline* *str*)

(setf *pt* (line-to-pt *testline*))
```