Part II
Chapter 3
An Editor

This chapter describes how a simple window-based text editor can be programmed in Clean. This chapter not only shows how such a program can be written in Clean, it also shows how in general window, keyboard and mouse actions can be defined.

3.1 The editor program state

The program state of the editor consists of a record of type `ProgState` with various kinds of information.

```
:: WndwFun .s .io :== (s,io) -> (s,io)

:: *ProgState =
/* Sect. 3.2 */{ name            :: String
/* Sect. 3.2 */, file            :: *File
/* Sect. 3.2 */, fileopened      :: Bool
/* Sect. 3.2 */, files           :: !Files
/* Sect. 3.2 */, lines           :: [String]

/* Sect. 3.4 */, startblinking   :: WndwFun ProgState (IOState ProgState)
/* Sect. 3.4 */, stopblinking    :: WndwFun ProgState (IOState ProgState)
/* Sect. 3.4 */, cursorvisible   :: Bool
/* Sect. 3.4 */, cursorpos       :: (Int,Int)     // x,y windowcoordinates

/* Sect. 3.2 */, clipboard       :: [String]
/* Sect. 3.2 */, selected        :: Bool
/* Sect. 3.2 */, selection       :: ((Int,Int),(Int,Int))
                                                      // from,to textcoordinates
/* Sect. 3.2 */, prevmouse       :: ButtonState
/* Sect. 3.2 */, font            :: InfoFont}

:: EDSIO :== (ProgState,IOState ProgState)
```

The use of the various fields will be explained in the corresponding sections.

Initialisation is done in a modular way assuring that an error will occur when a field is not properly initialised.

```
InitProgramState :: !Files -> ProgState;
InitProgramState fs
 = {     name            = abort ("name"          +++ errorstring),
         file            = abort ("file"          +++ errorstring),
         fileopened      = abort ("fileopened"    +++ errorstring),
         files           = fs,
         lines           = abort ("lines"         +++ errorstring),
```



```
            clipboard       = abort ("clipboard"    +++ errorstring),
            selected        = abort ("selected"     +++ errorstring),
            selection       = abort ("selection"    +++ errorstring),
            prevmouse       = abort ("prevmouse"    +++ errorstring),
            cursorpos       = abort ("cursorpos"    +++ errorstring),
            cursorvisible   = abort ("cursorvisible"+++ errorstring),
            font            = abort ("font"         +++ errorstring),
            startblinking   = abort ("startblinking"+++ errorstring),
            stopblinking    = abort ("stopblinking" +++ errorstring)
        }
    where
        errorstring = " field of state not properly initialized"

    Start world = closefiles finalfiles (CloseEvents restevents
    worldwithoutfilesandevents)
    where
        ({files=finalfiles},restevents)
          = StartIO devices (initialstate fs) [InitBlinker blinkTimerId] events
        (events,worldwithoutevents)     = OpenEvents world
        (fs,worldwithoutfilesandevents) = openfiles worldwithoutevents

        blinkTimerId    = inc DontCareId
        devices         = [MenuSystem [FileMenu,EditMenu,SearchMenu]]
        initialstate fs = seq [InitSaneMouse,InitCursor,InitHighlight,
                               InitFiles,InitFont,InitText,InitClipBoard]
                                                   (InitProgramState fs)
```

3.2 File access

The field `name` in the program state refers to the name of the file of which the text is being manipulated by the program. This name is used by the library functions for file I/O. It includes the path of directories or folders towards the file. Usually, when a file name is shown in a window this is done without the path. The function `stripDirs` provides the appropriate functionality.

```
stripDirs:: .a -> a | toString, fromString a
stripDirs namewithdirectories
  = toString (last (splitBy DirSeparator (fromString namewithdirectories)))
```

The functionality for reading files is the same as for the examples used in Chapter I.5. The filesystem is kept in the field files of the program state.

Similar functionality is used for writing files. An important aspect is that for writing to a file, that file must be unique. So, in the program state we have a field `file` of type `*File` to represent the file to be manipulated. Furthermore, there is a boolean `fileopened` indicating whether or not a file has been opened. This boolean is used to provide assistance to a user which attempts to perform operations wihtout opening a file.

When a file is opened its contents is read into the field `lines`: a list of strings representing the lines of the files. As an experiment we have chosen here to keep each string exactly as it is read as a line: including the newline character. This leads to a slight redundancy but it makes the code somewhat more straightforward at certain points.

The complete code for the file menu is given below. Note that since code from previous chapters is re-used it may be required to re-open a file for writing if it was opened for reading in an earlier stage.

```
InitFiles s = {s&name="",file=abort "No file opened",fileopened=False}

FileMenu = PullDownMenu DontCareId "File" Able
             [ MenuItem DontCareId "New..."    (Key 'N') Able New
             , MenuItem DontCareId "Open..."   (Key 'O') Able Open
             , MenuSeparator
             , MenuItem DontCareId "Close" (Key 'W') Able (OpenFirst Close)
             , MenuItem DontCareId "Save"  (Key 'S') Able (OpenFirst Save)
             ,MenuItem DontCareId "Save as..." NoKey Able (OpenFirst SaveAs)
```

```
            , MenuSeparator
            , MenuItem DontCareId "Quit"      (Key 'Q') Able Quit]

New s io
  = OutputFileDialog "Enter a name for the file:" "Untitled" NewFile (s,io)

Open s io = FileReadDialog (Show LineListRead FReadText) (s,io)

Quit state=:{fileopened=False} io = (state,QuitIO io)
Quit state /*fileopened=True*/ io = IOf QuitIO (Close state io)

Close state=:{file,files,fileopened} io // no check for saved
| not fileopened = (state,io)
| closeok        = (closedstate,CloseActiveWindow io)
| otherwise      = (closedstate,inform ["Error on Close",
                                        "Check your file system"]
                                                  (CloseActiveWindow io))
where
    (closeok,newfilesys) = fclose file files
    closedstate = {state&file=abort "no file opened after close",
                          selected=False,cursorpos=(0,0),
                           fileopened=False,files=newfilesys}

Save state=:{name} io = FileWrite name (state,io)

SaveAs s=:{name} io
 = OutputFileDialog "Save as:" name
          (\n sio-> ChangeFileNameInActiveWindowTitle (FileWrite n sio))
                                                                   (s,io)

OutputFileDialog message suggestedname fun (state=:{files},io)
| notcancel = fun fname ({state & files = nwfiles},nwio)
| otherwise = ({state & files = nwfiles},nwio)
where
    (notcancel,fname,nwfiles,nwio)  = SelectOutputFile message suggestedname
files io

ChangeFileNameInActiveWindowTitle (s=:{name},io) = (s,ChangeActiveWindowTitle
(stripDirs name) io)

FileWrite fname (s=:{files,file,name,lines},io)
| fname == name = ({s & files=nwfiles,file=LineListWrite lines (MakeWriteable
name file)},io)
| openok        = ({s & files=nwfiles,file=LineListWrite lines
writefile,name=fname},io)
| otherwise     = ({s & files=nwfiles},inform ["Could not open output file"] io)
where
    (openok, writefile,nwfiles) = fopen fname FWriteText files

MakeWriteable name readfile
| ok        = writefile
| otherwise = abort ("Could not write to file '"+++name+++"'")
where
    (ok, writefile) = freopen readfile FWriteText

LineListWrite:: [String] *File -> *File
LineListWrite []           f = f
LineListWrite [line:lines] f = LineListWrite lines (fwrites (toString line) f)

FileReadDialog fun (state=:{files},io)
| notcancel = fun {state & files = nwfiles, name = fname} nwio
| otherwise = ({state & files = nwfiles},nwio)
where
    (notcancel,fname,nwfiles,nwio)  = SelectInputFile files io

NewFile newfilename (s,io)  = seqIO [ Close
                                    , curry (Sf (\s-
>{s&name=newfilename,cursorpos=(0,0),selected=False}))
                                    , Show (\f->([],f)) FWriteText
                                    ] (s,io)
```



```
InitText s = {s& lines=[]}

Show readfun filemode s=:{name,files} io
| not readok = ({s&files=nwfiles},inform ["Could not open input file
'"+++name+++"'"] io)
| otherwise  = DisplayInWindow Close
({s&file=readf,fileopened=True,files=nwfiles,lines=linestext}, io)
where
    nonemptytext       = if (isEmpty text) [fromChar '\n'] text
    linestext          = if (lastline.[upb lastline] <> '\n') (init nonemptytext
++ [lastline+++"\n"]) nonemptytext
    lastline           = last nonemptytext
    (text,readf)       = readfun f
    (readok,f,nwfiles) = fopen name filemode files //FReadText files
```

3.3 Displaying the text

The text is displayed in a similar way as in Chapter I.5. Important differences are the blinking of the cursor (see Section ...), the mousefunction (Section ...) and the keyboardfunction (Section ...). Operations on the text use as much as possible the general update function to perform the appropriate operations on the screen. Clearly, most of the programming work is in the user interface. Performing the required operations on the list of strings is rather straightforward.

```
DisplayInWindow :: (WindowFunction ProgState (IOState ProgState))
                                                                 EDSIO -> EDSIO;
DisplayInWindow close (s=:{lines,name,font,startblinking},io)
        // lines must be non-empty, and always include a new-line character
    = startblinking (s,
            ChangeActiveWindowCursor IBeamCursor (OpenWindows [windef] io))
where
    windef = ScrollWindow DontCareId (0,0) (stripDirs name)
                           //id,position,title
              (ScrollBar (Thumb (~whiteMargin)) (Scroll font.width))
                                                                // horizontal
              (ScrollBar (Thumb 0) (Scroll font.height))    // vertical
              (toPictureDomain font (length lines)) (FontStringWidth (stripDirs
name) font.sysfont,font.height) MaxScrollWindowSize // active domain,minimum
size,init size
              updatefunction                          // window drawfunction
              [GoAway close,            // close file on close of the window
              Keyboard Able (getkeys True),         // keyboard handling
              Mouse Able (SaneMoveclickTrackMouse moveclickmouse trackmouse)]//
mouse handling


// domains are erased and all drawfuns are clipped when updatefunction is called
by the I/O lib
updatefunction domains s=:{selected,selection,cursorvisible,cursorpos,font,lines}
    = (s, HighlightUpdate font selected selection lines
       ++ CursorUpdate font cursorpos cursorvisible domains
       ++ TextUpdate font lines domains)

TextUpdate font lines domains = flatten (map update domains)
where
    update ((_,top),(_,bot)) = drawlines (tolinenumber font top) (tolinenumber
font (dec bot))

    drawlines first last = [ MovePenTo (0,(towindowcoordinate font first) +
font.up)
                           : flatten (map (drawline font) (lines%(first,last)))
                           ]

    drawline font line   = [DrawString line,MovePen (~(FontStringWidth line
font.sysfont),font.height)]

Update :: [((.Int,.Int),(.Int,.Int))] EDSIO -> EDSIO;
```

```
Update domains (s,io) = (ns,DrawInActiveWindow (map erasedomain domains ++
drawfuns) io)
where // does no clipping: highlight update

    erasedomain ((_,top),(_,bot)) = EraseRectangle
((~whiteMargin,top),(maxLineWidth,bot))

    (ns,drawfuns) = updatefunction domains s

UpdateScreen :: EDSIO -> EDSIO;
UpdateScreen (s,io) = Update [screen] (s,touchedio) // drawfuns outside
windowframe will be clipped
where
    (screen,touchedio) = ActiveWindowGetFrame io

UpdateScreenBelow font (_,yco) (s,io)
| downdrawingpos >= down = ScrollVertical (up + max font.height (pagesize / 2))
(s,touchedio)
| updrawingpos   <= up   = ScrollVertical (up - max font.height (pagesize / 2))
(s,touchedio)
| otherwise              = Update [((left,yco+font.height),(right,down))]
(s,touchedio)
where
    downdrawingpos = inc (yco + font.height)
    updrawingpos   = yco + font.up
    pagesize   = down - up
    (((left,up),(right,down)),touchedio) = ActiveWindowGetFrame io

UpdateThisLine oldpos newpos newreststring newlinestring lineindex (s=:{font},io)
    = seq ([ MoveCursor newpos
           , IOf (DrawInActiveWindow [ EraseRectangle (torestoflinerectangle
font newdrawpos)
                                     , MovePenToCursor font newdrawpos
                                     , DrawString newreststring])
           , Sf  (ChangeLine newlinestring lineindex)
           ]) (s,io)
where
    newdrawpos = if (beforeWinCo oldpos newpos) oldpos newpos


RemoveSelection :: EDSIO -> EDSIO;
RemoveSelection (s=:{selection=(fr,to),lines,font,startblinking},io)
    = seq [ Sf RemoveSelectedText
          , MoveCursor newcursorpos
          , startblinking
          , ScrollToMovedCursor
          , Update [torestoflinerectangle font newcursorpos]
          , UpdateScreenBelow font newcursorpos
          , AdaptPictureDomain
          ] (s,io)
where
    newcursorpos = towindowpos font lines fr

DoPaste :: EDSIO -> EDSIO;
DoPaste (s=:{cursorpos,clipboard,font},io)
     = seq [ Sf (\s->Insert clipboard s)
           , ScrollToMovedCursor
           , Update [torestoflinerectangle font cursorpos]
           , UpdateScreenBelow font cursorpos
           , AdaptPictureDomain
           ] (s,io)

TypeChar char (s,io)
| char == ReturnKey || char == EnterKey = TypeReturn (s,io)
| char == BackSpKey                     = TypeBackSpace (s,io)
| otherwise                             = TypeStdChar char (s,io)

TypeBackSpace (s=:{lines,cursorpos=cp=:(x,y),font},io)
| strindex > 0   = UpdateThisLine cp newpos newrestline newline lineindex (s,io)
| lineindex == 0 = (s,  Beep io)
```



```
| otherwise      = JoinBackwards (s,io)
where
    newpos = (x-FontCharWidth (line.[dec strindex]) font.sysfont,y)
    (_,(strindex,lineindex)) = MousePosToCursorPos font lines cp
    line        = lines!lineindex
    newrestline = line%(strindex,upb line)
    newline     = line%(0,strindex - 2)+++newrestline

JoinBackwards (s=:{lines,cursorpos,font},io)
    = seq ([ UpdateThisLine cursorpos newpos line joinedline prevlineindex
           , Sf  (DeleteLine lineindex)
           , UpdateScreenBelow font newpos
           , AdaptPictureDomain
           ])  (s,io)
where
    newpos = (FontStringWidth prevline font.sysfont,towindowcoordinate font
prevlineindex)
    (_,(_,lineindex)) = MousePosToCursorPos font lines cursorpos
    line              = lines!lineindex
    prevlineindex     = dec lineindex
    prevline          = lines!prevlineindex
    joinedline        = prevline%(0,dec (upb prevline))+++line

TypeReturn (s=:{lines,cursorpos=cp=:(x,y),font},io)
    = seq ([ UpdateThisLine cp newpos "" changeline lineindex
           , Sf  (InsertLine insertline newlineindex)
           , AdaptPictureDomain
           , UpdateScreenBelow font (0,towindowcoordinate font lineindex)
           ])  (s,io)
where
    newpos = (0,towindowcoordinate font newlineindex)
    (_,(strindex,lineindex)) = MousePosToCursorPos font lines cp
    line         = lines!lineindex
    newlineindex = inc lineindex
    changeline   = line%(0,dec strindex)+++toString ReturnKey
    insertline   = line%(strindex,upb line)

TypeStdChar char (s=:{lines,cursorpos=cp=:(x,y),font},io) //cp needed due to bug
in syntax
    = UpdateThisLine cp newpos newrestline newline lineindex (s,io)
where
    newpos = (x+FontCharWidth char font.sysfont,y)
    (_,(strindex,lineindex)) = MousePosToCursorPos font lines cp
    line        = lines!lineindex
    newrestline = toString char+++line%(strindex,upb line)
    newline     = line%(0,dec strindex)+++newrestline

ChangeLine newline index s=:{lines} = {s&lines=updateAt index newline lines}

InsertLine newline index s=:{lines} = {s&lines=insertAt index newline lines}

Insert :: [{#.Char}] ProgState -> ProgState
Insert text s=:{font,lines,cursorpos}
| text == [] = s
| otherwise  = DoInsert (frompos,fromlineindex) (thisline%(frompos,upb thisline))
text s
where
    thisline = lines!fromlineindex
    (_,(frompos,fromlineindex)) = MousePosToCursorPos font lines cursorpos

    DoInsert (frompos,fromlineindex) oldrestline [] s = InsertLine oldrestline
fromlineindex s
    DoInsert (frompos,fromlineindex) oldrestline [onestring] s=:{lines}
    | onestring.[upb onestring] == '\n' = seq [ChangeLine newfromline
fromlineindex,
                                               InsertLine oldrestline (inc
fromlineindex)] s
    | otherwise                         = ChangeLine (newfromline+++oldrestline)
fromlineindex s
    where
```

```
        thislline    = lines!fromlineindex
        newfromline
        | frompos == 0 = onestring
        | otherwise    = thisline%(0,dec frompos)+++onestring

    DoInsert (frompos,fromlineindex) oldrestline [firstline:rest] s=:{lines}
                                        = seq (flatten
                                              [[ChangeLine newfromline
fromlineindex],
                                               [InsertLine line index \\ line <-
init rest & index <-[inc fromlineindex..]],
                                               [InsertLine (last
rest+++oldrestline) (fromlineindex+length rest)]
                                              ]) s
    where
        thisline    = lines!fromlineindex
        newfromline
        | frompos == 0 = firstline
        | otherwise    = thisline%(0,dec frompos)+++firstline

DeleteLine         index s=:{lines} = {s&lines=removeAt index lines}

DeleteLines (fromindex,toindex) s=:{lines} = {s&lines=removeelements
(fromindex,toindex) lines}

RemoveSelectedText s=:{selection=:((frompos,fromline),(topos,toline)),lines}
    = seq [DeleteLines (fromline,toline),InsertLine (frompart+++topart)
fromline] {s&selected=False}
where
    frompart = (if (frompos == 0) "" (lines!fromline%(0,dec frompos)))
    topart   = lines!toline%(topos,upb (lines!toline))
```

3.4 Cursor handling

The cursor is displayed by a function that is invoked by a timer. The functions to start and stop blinking (necessary when selections are made) are put in the state (in the fields startblinking and stopblinking). This makes it possible to define the id of the timer locally where the function InitBlinker is called. This restricts the use of this id to the expression where InitBlinker is called (StartIO).

Certain operations require that the cursor is moved. That operations need to know the cursorsp position and whether the cursor is visible at that moment (fields of the state: cursorpos and cursorvisible). At certain points the mouse-coordinates of the cursor have to be converted to text-coordinates and vice-versa (see Section ...).

```
InitCursor :: ProgState -> ProgState
InitCursor s = {s&cursorpos=(0,0),cursorvisible=False}

CursorUpdate :: .InfoFont (.Int,.Int) .Bool [((.Int,.Int),(.Int,.Int))]->
[.(Picture -> Picture)]
CursorUpdate font cursorpos cursorvisible domains
| cursorvisible && mustbeupdated = [DrawCursor font cursorpos]
| otherwise     = []
where
    mustbeupdated = or (map (inside cursorpos) domains)

    inside cursorpos domain = or [insideDomain (x,y) domain \\ y <-
[yup..ydown]]

    ((x,yup),(_,ydown)) = CursorDrawArea font cursorpos

InitBlinker :: .TimerId ProgState (IOState ProgState) -> EDSIO
InitBlinker blinkTimerId s io
| blink > 0 =  (InitBlinkState s, OpenTimer (Timer blinkTimerId Unable blink
BlinkCursor) touchedio)
| otherwise =  (InitBlinkState s, touchedio)
where
```



```
    (blink,touchedio) = GetTimerBlinkInterval io

    InitBlinkState s = {s&startblinking=StartBlinking,stopblinking=StopBlinking}

    StopBlinking  (s,io) = (s,DisableTimer blinkTimerId io)

    StartBlinking (s,io) = (s,EnableTimer blinkTimerId io)

BlinkCursor :: .a ProgState (IOState *b) -> (ProgState,IOState *b)
BlinkCursor nrintervalspassed  s=:{cursorpos,cursorvisible=oldvisibility,font} io
   = ({s&cursorvisible=not oldvisibility},DrawInActiveWindow [DrawCursor font
cursorpos] io)

DrawCursor font cursorpos pic
//   = SetPenMode CopyMode (DrawLine ((x,y+font.lead),(x,y+dec font.height))
(SetPenMode XorMode pic))
   = seq [SetPenMode XorMode,DrawLine (CursorDrawArea font cursorpos),SetPenMode
CopyMode] pic

CursorDrawArea font (x,y) = ((x,y+font.lead),(x,y+dec font.height))

MoveCursor :: (.Int,.Int) (ProgState,IOState *a) -> (ProgState,IOState *a)
MoveCursor newpos (s,io) = SetCursorpos newpos (RemoveCursor (s,io))

MoveCursorLeft :: (ProgState,IOState *a) -> (ProgState,IOState *a)
MoveCursorLeft (s=:{font,lines,cursorpos=:cursorpos=:(xpos,ypos)},io)
| strindex == 0 && linenumber == 0 = (s,Beep io)
| strindex == 0 = MoveCursor (towindowpos font lines (size (lines!(dec
linenumber)),dec linenumber)) (s,io)
| otherwise     = MoveCursor (towindowpos font lines (strindex-1,linenumber))
(s,io)
where
  (_,(strindex,linenumber)) = MousePosToCursorPos font lines cursorpos

MoveCursorRight :: Bool (ProgState,IOState *a) -> (ProgState,IOState *a)
MoveCursorRight beep (s=:{font,lines,cursorpos=:cursorpos=:(xpos,ypos)},io)
| strindex == upbline && linenumber == dec (length lines) && beep = (s,Beep io)
| strindex == upbline && linenumber == dec (length lines) = (s,io)
| strindex == upbline = MoveCursor (towindowpos font lines (0,inc linenumber))
(s,io)
| otherwise           = MoveCursor (towindowpos font lines (inc
strindex,linenumber)) (s,io)
where
  (_,(strindex,linenumber)) = MousePosToCursorPos font lines cursorpos
  upbline  = dec (size (lines!linenumber))

MoveCursorUp :: (ProgState,IOState *a) -> (ProgState,IOState *a)
MoveCursorUp (s=:{font,lines,cursorpos=:cursorpos=:(xpos,_)},io)
| linenumber == 0 = (s,Beep io)
| otherwise       = MoveCursor (towindowpos font lines (newstrindex,dec
linenumber)) (s,io)
where
  (_,(newstrindex,_)) = MousePosToCursorPos font lines (xpos,towindowcoordinate
font (dec linenumber))
  (_,(_,linenumber))  = MousePosToCursorPos font lines cursorpos

MoveCursorDown :: (ProgState,IOState *a) -> (ProgState,IOState *a)
MoveCursorDown (s=:{font,lines,cursorpos=:cursorpos=:(xpos,_)},io)
| linenumber == dec (length lines) = (s,Beep io)
| otherwise = MoveCursor (towindowpos font lines (newstrindex,inc linenumber))
(s,io)
where
  (_,(newstrindex,_)) = MousePosToCursorPos font lines (xpos,towindowcoordinate
font (inc linenumber))
  (_,(_,linenumber))  = MousePosToCursorPos font lines cursorpos

RemoveCursor (s=:{cursorpos,cursorvisible,font},io)
| cursorvisible = ({s&cursorvisible=False},DrawInActiveWindow [DrawCursor font
cursorpos] io)
| otherwise     = (s,io)
```

```
SetCursorpos newpos (s,io) = ({s&cursorpos=newpos},io)

MovePenToCursor :: .InfoFont (.Int,.Int) -> .(Picture -> Picture)
MovePenToCursor font (x,y) = MovePenTo (x,y+font.up)

ScrollToStart :: EDSIO -> EDSIO
ScrollToStart (s=:{font,lines},io)
    = ChangeActiveScrollBar (ChangeThumbs left up) s io
where
    ((left,up),_) = toPictureDomain font (length lines)

ScrollToEnd :: EDSIO -> EDSIO
ScrollToEnd (s=:{font,lines},io) =
     ChangeActiveScrollBar (ChangeThumbs left down) s io
where
    ((left,_),(_,down)) = toPictureDomain font (length lines)

ScrollIfOutOfFrame::(((Int,Int),(Int,Int))->(Int,Int,Int,Int)) EDSIO->EDSIO
ScrollIfOutOfFrame calcnewthumbs (s=:{cursorpos=(x,y),font},io)
| x < horthumb                  = ScrollHorizontal newleft  (s,io1)
| x >= horright             = ScrollHorizontal newright (s,io1)
| y < verthumb                  = ScrollVertical   newup    (s,io1)
| y + font.height >= verdown    = ScrollVertical   newdown  (s,io1)
| otherwise                     = (s,io1)
where
    (frame=:((horthumb,verthumb),(horright,verdown)),io1)
                                        = ActiveWindowGetFrame io
    (newleft,newright,newup,newdown)    = calcnewthumbs frame

ScrollSlowlyWithMovingCursor :: EDSIO -> EDSIO
ScrollSlowlyWithMovingCursor (s=:{font},io)
    = ScrollIfOutOfFrame (move font.height) (s,io)
where
    move offset ((horthumb,verthumb),(horright,verdown))
        = (horthumb - offset,horthumb + offset,
            verthumb - offset,verthumb + offset)

ScrollWithMovingCursor ::EDSIO -> EDSIO
ScrollWithMovingCursor (s=:{font},io)
    = ScrollIfOutOfFrame (movehalfpage font.height) (s,io)
where
    movehalfpage offset ((horthumb,verthumb),(horright,verdown))
        = (horthumb - horoffset,horthumb + horoffset,
            verthumb - veroffset,verthumb + veroffset)
    where
        horoffset  = max font.height (pagewidth/2)
        veroffset  = max font.height (pagelength/2)
        pagelength = verdown  - verthumb
        pagewidth  = horright - horthumb

ScrollToMovedCursor :: EDSIO -> EDSIO
ScrollToMovedCursor (s=:{cursorpos},io)
    = ScrollIfOutOfFrame (topos cursorpos) (s,io)
where
    topos (x,y) frame = (x,x,y,y)

MousePosToCursorPos::.InfoFont .[{#Char}] (.Int,.Int)->
                                                (.(Int,Int),.(Int,Int))
MousePosToCursorPos font lines (xco,yco) // no corrections (yet) for being
horizontally outside the domain
| yco < 0  = ((0,0),(0,0))
| lineindex >= length lines
            = ( (FontStringWidth (lines!maxindex) font.sysfont,
                                towindowcoordinate font maxindex),
                (upb (lines!maxindex),maxindex))
| otherwise = ((xpos,ypos),(strindex,lineindex))
where
    ypos            = towindowcoordinate font lineindex
    lineindex        = tolinenumber font yco
```



```
    linestring      = lines!lineindex
    lengthstring    = size linestring
    (strindex,xpos) = findpos (0,0) // ->(pos in string,windowcoordinate)
    maxindex        = dec (length lines)

    findpos thispos=:(thisindex,thisco)
    | thisindex == dec lengthstring = thispos
    | nextco >= xco  = if (nextco - xco < xco - thisco) nextpos thispos
    | otherwise      = findpos nextpos
    where
        nextpos     = (inc thisindex, nextco)

        nextco
        | thisindex < lengthstring
              = thisco + FontCharWidth linestring.[thisindex] font.sysfont
        | otherwise = maxLineWidth
```

3.5 Text and window coordinates

```
tolinenumber :: !.InfoFont !.Int -> Int;
tolinenumber font windowcoordinate
| windowcoordinate > 0 = windowcoordinate / font.height
| otherwise            = 0

towindowcoordinate :: !.InfoFont !.Int -> Int;
towindowcoordinate font linenumber = linenumber * font.height // top of the line

whiteMargin :: .Int;
whiteMargin  = 5

maxLineWidth :: .Int;
maxLineWidth = 1024

toPictureDomain :: .InfoFont .Int -> ((Int,Int),(Int,Int))
toPictureDomain font nrlines = ((~whiteMargin,0),maxdom MaxScrollWindowSize
(maxLineWidth,nrlines*font.height))
where
    maxdom x y
    | beforeWinCo x y = y
    | otherwise       = x

towindowrectangle :: .InfoFont .Int -> (.(Int,Int),(.Int,Int));
towindowrectangle font linenumber = ((0,winco),(maxLineWidth,winco +
font.height)) // surrounding rectangle
where
    winco = towindowcoordinate font linenumber

towindowpos :: .InfoFont [.{#Char}] !(.Int,.Int) -> .(Int,Int);
towindowpos font lines (xco,yco) = (towindowx font (lines!yco)
xco,towindowcoordinate font yco)

towindowx :: .InfoFont .{#Char} !.Int -> Int;
towindowx font line xco
| xco == 0  = 0
| otherwise = FontStringWidth (line%(0,dec xco)) font.sysfont

torestoflinerectangle :: .InfoFont !u:(.a,Int) -> (v:(.a,Int),(.Int,Int)), [u <=
v];
torestoflinerectangle font cursorpos=:(_,yco) =
(cursorpos,(maxLineWidth,yco+font.height))

AdaptPictureDomain :: EDSIO -> EDSIO
AdaptPictureDomain (s=:{font,lines},io) = ChangeActivePictureDomain
(toPictureDomain font (length lines)) s io
```

3.6 Editing with keyboard actions

```
EditMenu = PullDownMenu DontCareId "Edit" Able
```

```
            [ MenuItem DontCareId "Undo"  (Key 'Z') Unable NotImplemented
            , MenuSeparator
            , MenuItem DontCareId "Cut"   (Key 'X') Able (SelectFirst Cut)
            , MenuItem DontCareId "Copy"  (Key 'C') Able (SelectFirst Copy)
            , MenuItem DontCareId "Paste" (Key 'V') Able Paste
            , MenuItem DontCareId "Clear" NoKey     Able Clear
            , MenuSeparator
            , MenuItem DontCareId "Balance"    (Key 'B') Able NotImplemented
            , MenuItem DontCareId "SelectAll"  (Key 'A') Able (OpenFirst
SelectAll)
            , MenuSeparator
            , MenuItem DontCareId "Format..."  (Key 'J') Able (OpenFirst
Format)]

SearchMenu = PullDownMenu DontCareId "Search" Able
            [ MenuItem DontCareId "Find..."      (Key 'F') Able NotImplemented
//(OpenFirst Find)
            , MenuItem DontCareId "Find Next"    (Key 'G') Able NotImplemented
            , MenuSeparator
            , MenuItem DontCareId "Goto Line..." NoKey Able (OpenFirst
GotoLine)]

NotImplemented s io = (s,inform ["This Function is not (yet) implemented"] io)

OpenFirst f s=:{fileopened} io
| not fileopened = (s,inform ["Please, first open a file."] io)
| otherwise      = f s io

SelectFirst f s=:{selected} io
| not selected   = (s,inform ["Please, first make a selection."] io)
| otherwise      = f s io
/*
Find s io = inputdialog "Find" (MM 50.0) findstring s io
where
    findstring string s=:{lines} io
     =  case searchAll_RE (fromString string) (flatten (map fromString lines))
of
         PFound [([((_,(firstchar,lastchar)),_):_],_):_]
               -> Select (toselection firstchar (lastchar-firstchar) lines
(0,0)) (s,io)
//             -> warnOK (firstchar,lastchar) (curry (Select (toselection
firstchar (lastchar-firstchar) lines (0,0)))) s io
         else  -> (s, Beep io)

    toselection charnr nrchars []           fr = (fr,fr)
    toselection charnr nrchars [line:lines] (fromx,fromy)
    | upb line < charnr = toselection (charnr-size line) nrchars lines (0,inc
fromy)
    | otherwise         = ((charnr,fromy),taketo nrchars [line%(charnr,upb
line):lines] (charnr,fromy))

    taketo nrchars []           (tox,toy) = (tox,toy)
    taketo nrchars [line:lines] (tox,toy)
    | upb line < nrchars = taketo (nrchars - size line) lines (0,inc toy)
    | otherwise          = (nrchars + tox,toy)
*/
InitClipBoard  s = {s&clipboard=[]}
ClearClipBoard s = {s&clipboard=[]}

Cut s io = RemoveSelection (Copy s io)

//Paste s=:{clipboard} io = seqIO [getkeys True
(key,KeyDown,(False,False,False,False)) \\ key <- flatten (map fromString
clipboard)] (s,io)

Paste s=:{selected=True} io = seq [ RemoveSelection, DoPaste] (s,io)
Paste s/*selected=False*/io = DoPaste (s,io)

Clear s=:{clipboard} io      = (ClearClipBoard s,io)
```



```
Copy s=:{selection,lines} io = ({s&clipboard=copy selection lines},io)
where
    copy ((frx,frline),(tox,toline)) lines
     | frline == toline && frx == tox  = []
     | frline == toline                = [(lines!frline)%(frx,dec tox)]
     | otherwise                       = [(lines!frline)%(frx,upb
(lines!frline))] ++ copy ((0,inc frline),(tox,toline)) lines

Select sel (s=:{selected,stopblinking},io)
| selected  = seq [ RemoveHighlight, DoSelect sel] (s,io)
| otherwise = seq [ stopblinking,    DoSelect sel] (s,io)
where
   DoSelect sel=:(fr,to) (s=:{font,lines},io)
    = seq [ Sf (\s->{s&selected=True,selection=sel})
          , MoveCursor (towindowpos font lines fr)
          , Highlight
          , ScrollToMovedCursor
          ] (s,io)

SelectAll s=:{lines} io = Select ((0,0),(upb (lines!lastindex),lastindex)) (s,io)
where
    lastindex = dec (length lines)

GotoLine s io = inputdialog "Goto Line" (MM 50.0) (\inputstring->GotoLineNr
(fromString inputstring)) s io
where
    GotoLineNr linenr s=:{font,lines} io
     | linenr >= length lines = (s,Beep io)
     | otherwise              = ScrollToMovedCursor (MoveCursor
(0,towindowcoordinate font linenr) (s,io))


getkeys beep (key,kstate,modifs=:(shift,opt,comm,contr)) s=:{selected} io
| kstate == KeyUp  = (s,io)
| key == BeginKey  = ScrollToStart  (s,io)
| key == EndKey    = ScrollToEnd    (s,io)
| key == PgUpKey   = ScrollPageUp   (s,io)
| key == PgDownKey = ScrollPageDown (s,io)
| key == LeftKey   = ScrollWithMovingCursor (MoveCursorLeft  (s,io))
| key == RightKey  = ScrollWithMovingCursor (MoveCursorRight beep (s,io))
| key == UpKey     = ScrollWithMovingCursor (MoveCursorUp    (s,io))
| key == DownKey   = ScrollWithMovingCursor (MoveCursorDown  (s,io))
| (key == BackSpKey || key == DelKey) && selected = RemoveSelection (s,io)
| key == DelKey    = seqIO [getkeys (not beep) (RightKey,kstate,modifs),
                            getkeys beep (BackSpKey,kstate,modifs)] (s,io)
| selected         = TypeChar key (RemoveSelection (s,io))
| otherwise        = TypeChar key (s,io)
```

3.7 Highlighting

For high lighting it is important to know whether or not something is selected and which part of the text is selected (fields in the state selected, selection). The selection coordinates are text coordinates that have to be converted to window coordinates and vice-versa. The screen effects of highlighting and de-highlighting are performed by one and the same function which produces a list of screen draw functions that is its own reverse (assuming that the default mode of drawing is copy mode). These functions set the screen in hilitemode, fill the required rectangles and reset the screen to copy mode again.

```
InitHighlight   :: ProgState -> ProgState
InitHighlight s = {s&selected=False,selection=((0,0),(0,0))}

HighlightUpdate :: .InfoFont .Bool ((Int,Int),(Int,Int)) [{#.Char}] -> .[Picture
-> Picture] // textcoordinates
HighlightUpdate font selected (fr=:(_,frline),to=:(_,toline)) lines
| beforeWinCo to fr = HighlightUpdate font selected (to,fr) lines
| selected          = HighlightDrawFuns (selrectangles font (frline,toline) sel)
```

```
| otherwise          = []
where
    sel = (towindowpos font lines fr,towindowpos font lines to)

RemoveHighlight :: (ProgState,IOState *a) -> (ProgState,IOState *a)
RemoveHighlight (s,io) = seq [Highlight, Sf (\s->{s&selected=False})] (s,io)

Highlight :: (ProgState,IOState *a) -> (ProgState,IOState *a)
Highlight (s=:{selection,font,lines},io)
    = (s,DrawInActiveWindow (HighlightUpdate font True selection lines) io)

DrawHighlight   :: .InfoFont ((Int,Int),(Int,Int)) -> .[Picture -> Picture] //
windowcoordinates
DrawHighlight font selection=:((_,fromy),(_,toy))
    = HighlightDrawFuns (selrectangles font (frline,toline) selection)
where
    frline   = tolinenumber font fromy
    toline   = tolinenumber font toy

selrectangles font (frline,toline) (fr=:(fromx,fromy),to=:(tox,toy)) //
textcoordinates, windowcoordinates
| fr == to         = []
| frline == toline = if (fromx < tox) [((fromx,frrectco),(tox,torectco))]
                                      [((tox,frrectco),(fromx,torectco))]
| frline < toline  = [((fromx,frrectco),(maxLineWidth,towindowcoordinate font
(inc frline)))] ++
                     map (towindowrectangle font) [inc frline..dec toline] ++
                     [((0,towindowcoordinate font toline),(tox,torectco))]
| otherwise        = selrectangles font (toline,frline) (to,fr)
where
    frrectco = towindowcoordinate font frline
    torectco = towindowcoordinate font (inc toline)

HighlightDrawFuns rectangles =    [SetPenMode HiliteMode]
                               ++ map FillRectangle rectangles
                               ++ [SetPenMode CopyMode]
```

3.8 'Sane' mouse handling

```
InitSaneMouse :: !ProgState -> ProgState;
InitSaneMouse s = {s&prevmouse=ButtonUp}

PrevMouse button s io = ({s&prevmouse=button},io)

:: MouseFun :== MouseFunction ProgState (IOState ProgState)

SaneMouse:: MouseFun MouseState ProgState (IOState ProgState) -> EDSIO
SaneMouse mousefun mouseinfo=:(pos,button,modifiers) s=:{prevmouse} io
| button==ButtonStillDown && prevmouse==ButtonUp = seqIO [ PrevMouse button
                                                         , mousefun
(pos,ButtonDown,modifiers)
                                                         , mousefun mouseinfo
                                                         ] (s,io)
| IsClick button && IsClick prevmouse            = seqIO [ PrevMouse button
                                                         , mousefun
(pos,ButtonUp,modifiers)
                                                         , mousefun mouseinfo
                                                         ] (s,io)
| IsClick button && prevmouse==ButtonStillDown   = seqIO [ PrevMouse button
                                                         , mousefun
(pos,ButtonUp,modifiers)
                                                         , mousefun mouseinfo
                                                         ] (s,io)
| otherwise                                      = seqIO [ PrevMouse button
                                                         , mousefun mouseinfo
                                                         ] (s,io)

SaneMoveclickTrackMouse :: MouseFun MouseFun MouseState ProgState (IOState
ProgState) -> EDSIO
```



```
SaneMoveclickTrackMouse moveclickfun trackfun mouseinfo=:(pos,button,modifiers)
s=:{prevmouse} io
| button==ButtonStillDown && prevmouse==ButtonUp = seqIO [ PrevMouse button
                                                         , moveclickfun
(pos,ButtonDown,modifiers)
                                                         , moveclickfun
mouseinfo
                                                         , SwitchToTracking
trackfun moveclickfun
                                                         ] (s,io)
| button==ButtonStillDown                        = seqIO [ PrevMouse button
                                                         , moveclickfun
mouseinfo
                                                         , SwitchToTracking
trackfun moveclickfun
                                                         ] (s,io)
| IsClick button && IsClick prevmouse            = seqIO [ PrevMouse button
                                                         , moveclickfun
(pos,ButtonUp,modifiers)
                                                         , moveclickfun mouseinfo
                                                         ] (s,io)
| otherwise                                      = seqIO [ PrevMouse button
                                                         , moveclickfun
mouseinfo
                                                         ] (s,io)

SaneTrackMouse trackfun moveclickfun mouseinfo=:(pos,button,modifiers) s io
| button == ButtonStillDown = trackfun mouseinfo s io
| button == ButtonUp        = seqIO [ PrevMouse ButtonUp
                                    , trackfun mouseinfo                //
finish tracking
                                    , SwitchToMoveclicking moveclickfun trackfun
                                    ] (s,io)
| otherwise                 = seqIO [ PrevMouse button
                                    , trackfun (pos,ButtonUp,modifiers) //
Insert ButtonUp: finish tracking
                                    , moveclickfun mouseinfo
                                    , SwitchToMoveclicking moveclickfun
trackfun] (s,io)

SwitchToTracking trackfun moveclickfun s io
    = (s,ChangeActiveMouseFunction (SaneTrackMouse trackfun moveclickfun) io)

SwitchToMoveclicking moveclickfun trackfun s io
    = (s,ChangeActiveMouseFunction (SaneMoveclickTrackMouse moveclickfun
trackfun) io)

moveclickmouse mouseinfo=:(pos,buttonstate,modifs=:(shift,opt,comm,contr)) s io
| IsClick buttonstate             = clickmouse pos s io
| buttonstate == ButtonStillDown = endclicking s io
| otherwise /*ButtonUp*/          = (s,io)

clickmouse mousepos s=:{selected,lines,font,startblinking} io
| selected  = seq [ MoveCursor newcursorpos, RemoveHighlight, startblinking ]
(s,io)
| otherwise =       MoveCursor newcursorpos (s,io)
where
    (newcursorpos,_) = MousePosToCursorPos font lines mousepos

endclicking s=:{stopblinking,cursorpos,font,lines} io
    = seq [stopblinking, Sf (\s-
>{s&selected=True,selection=(begintrackpos,begintrackpos)})] (s,io)
where
    (_,begintrackpos) = MousePosToCursorPos font lines cursorpos


trackmouse (pos,ButtonStillDown,_) s=:{font,lines} io  = showtracking pos s io
trackmouse _  /*ButtonUp*/         s io = endtracking s io
```

```
showtracking mousepos=:(xco,yco)
s=:{selection=(begintrackpos,_),cursorpos,font,lines} io
| oldtextpos == newtextpos = ScrollSlowlyWithMovingCursor (s,io) // if not
already there...
| otherwise  = seq [ MoveCursor newcursorpos
                   , IOf (DrawInActiveWindow (DrawHighlight font
(cursorpos,newcursorpos)))
                   , Sf (\s->{s&selection=(begintrackpos,newtextpos)})

                   , ScrollSlowlyWithMovingCursor
                   ] (s,io)
where
    (newcursorpos,newtextpos) = MousePosToCursorPos font lines mousepos
    (_,oldtextpos)            = MousePosToCursorPos font lines cursorpos

endtracking s=:{selection=(begintrackpos,_),font,lines,cursorpos,startblinking}
io
| begintrackpos == cursortextpos = startblinking ({s&selected=False},io)
| otherwise                      = ({s&selection=orderWinCo
(begintrackpos,cursortextpos)},io)
where
    (_,cursortextpos) = MousePosToCursorPos font lines cursorpos

//getmouse (pos=:(_,y),ButtonDoubleDown,_) s=:{stopblinking,font} io
//  = seq [ stopblinking, MoveCursor pos, Highlightline (tolinenumber font y) ]
(s,io)
```

3.9 Changing Font

```
InitFont :: !ProgState -> ProgState
InitFont s = {s&font=DefaultInfoFont}


Format s=:{font} io =  FontSel UpdateAndFontChange font s io
where
    UpdateAndFontChange newfont (s,io)
        = UpdateAfterFontChange ({s&font=newfont},
                                 DrawInActiveWindow [SetFontName
newfont.fontname,
                                                     SetFontSize
newfont.fontsize] io)

    UpdateAfterFontChange (s=:{font,lines,cursorpos},io)
        = seq [AdaptPictureDomain,
               uncurry (ChangeActiveScrollBar (ChangeScrolls font.width
font.height)),
               MoveCursor newcursorpos,
               UpdateScreen] (s,io)
    where
        (newcursorpos,_) = MousePosToCursorPos font lines cursorpos

FontSel :: (InfoFont -> .((*a,IOState *a) -> (*a,IOState *a))) .InfoFont *a
(IOState *a) -> (*a,IOState *a)
FontSel action font s io
 = OpenModalDialog (CommandDialog DontCareId "Change Font" [ItemSpace (Pixel 12)
(Pixel 6)] okId
        [ StaticText ftextId Left "Font:"
        , ScrollingList scrollfnameId (XOffset ftextId (MM 1.0)) (MM 0.0) Able 5
                                      font.fontname names adaptfoxandshowsizes
        , StaticText fsizetextId (RightTo scrollfnameId) "Size:"
        , ScrollingList scrollfsizeId (XOffset fsizetextId (MM 1.0)) (MM 0.0)
Able 5
                                      (toString font.fontsize) (map toString
(FontSizes font.fontname)) adaptfox
        , DialogIconButton foxId Left ((0,0),(dialogwidth,2*foxheight)) (fox
font) Unable buttondummy
        , DialogButton cancelId Left "Cancel" Able nofontchange
```



```
        , DialogButton okId (XOffset cancelId (Pixel (dialogwidth - 140))) "OK"
Able (fontchange action)]) s io
where
    dialogwidth = 260

    names = map tostring (sort (map tochars FontNames))

    tostring [] = ""
    tostring [x:xs] = toString x +++ tostring xs

    tochars:: String -> [Char]
    tochars s = fromString s

    [ftextId,scrollfnameId,fsizetextId,scrollfsizeId,foxId,cancelId,okId:_] =
[inc DontCareId..]

    nofontchange dinfo s io = (s,CloseActiveDialog io)

    fontchange action dinfo s io  = action newfont (s,CloseActiveDialog io)
    where
        newfont  = MakeInfoFont (fontname,[],fontsize)
        fontname = GetScrollingListItem scrollfnameId dinfo
        fontsize = fromString (GetScrollingListItem scrollfsizeId dinfo)

    fox {sysfont,ascent,height} buttonselectstate
        = [ SetFont sysfont
          , MovePenTo (0,max ascent (foxheight/2))
          , DrawString firstline
          , MovePen (~(FontStringWidth firstline sysfont),height)
          , DrawString    "jumps over the lazy dog"]
    where
        firstline = "The quick brown fox"

    adaptfox  dinfo dstate = ChangeIconLook foxId (fox (MakeInfoFont
(fontname,[],selsize))) dstate
    where
        fontname    = GetScrollingListItem scrollfnameId dinfo
        selsize     = fit oldfontsize (FontSizes fontname)
        oldfontsize = fromString (GetScrollingListItem scrollfsizeId dinfo)

    adaptfoxandshowsizes dinfo dstate
        = seq [ ChangeScrollingList scrollfsizeId (toString selsize) (map
toString fontsizes)
              , ChangeIconLook foxId (fox (MakeInfoFont (fontname,[],selsize)))]
dstate
    where
        oldfontsize = fromString (GetScrollingListItem scrollfsizeId dinfo)
        fontname    = GetScrollingListItem scrollfnameId dinfo
        selsize     = fit oldfontsize fontsizes
        fontsizes   = FontSizes fontname

    fit nr []     = zero
    fit nr [x]    = x
    fit nr [x,y:ys]
     | nr <= x          = x
     | nr > x && nr < y = if (nr - x < y - nr) x y
     | otherwise        = fit nr [y:ys]

    foxheight = 32
```

3.10 Exercises