```
(:= *build-module-list* '(

    utilities:sharp-sharp

    drawing:dag
    drawing:dag-input-node
    drawing:dag-util
    drawing:dag-link
    drawing:dag-coords
    drawing:dag-impress
    drawing:impress
    drawing:impress-textures
    drawing:impress-draw
    ) )

(prompt (initprompt (chrval "$") ) )

(build *build-module-list*)
```

```
(include drawing:dag-declarations)

(def-struct dag:node
   (type               'node)
    name
   (parents            () suppress)
   (mommy              () suppress)
   (children           () suppress)
   (child-edge-styles  () suppress)
    depth
    height
    column
    x-coord
    y-coord
   (delta-x            0)
   (flag               0)
   (links              () suppress)
   (left               () suppress)
   (right              () suppress)
    label
    style
    )


(:= *dag:left*  (dag:node:new) )
(:= *dag:right* (dag:node:new) )
(:= (dag:node:links *dag:left*)  '(,*dag:right*) )
(:= (dag:node:links *dag:right*) '(,*dag:left*) )


(defun dag:impress ( &optional (in-file "in.dag") (out-file "dag.impa") )
    (unwind-protect
        (let ()
            ;(time-functions 'dag:read-graph
            ;                'dag:sort-graph
            ;                'dag:set-x-coords
            ;                'dag:set-y-coords
            ;                'dag:impress-graph)

            (dag:read-graph in-file)
            (dag:sort-graph)
            (dag:set-x-coords)
            (dag:set-y-coords)
            (dag:impress-graph out-file)
            out-file)
        ;(print-function-times)
        ;(untime-functions)
        out-file) )


(defun dag:sort-level (level)
    (loop (initial nodes ()
                   width 0
                   last  *dag:left*
                   left  ()
                   right (for (elt in level)
                              (splice (dag:copy elt) ) ) )
          (while level)
          (next nodes (pop level) )
          (do (for (node in nodes)
                   (do (pop right) ) )
```



```
              (:= nodes (dag:sort-list nodes left right) )
              (for (node in nodes)
                   (do (:= width (+ width (dag:node-width node) ) )
                       (:= (dag:node:left  node) last)
                       (:= (dag:node:right last) node)
                       (:= last node)
                       (push left node) ) )
              (:= (dag:node:right last) *dag:right*) )
          (result (push *dag:widths* width)
                  (push *dag:edges* (dag:node:right *dag:left*) )
                  (reverse left) ) ) )

(defun dag:sort-list (nodes left right)
    (if (! (cdr nodes) )
        nodes
        (dag:real-sort-list nodes left right) ) )

(defun dag:real-sort-list (nodes left right)
    (:= (dag:node:links *dag:left*)  '(,*dag:right*) )
    (:= (dag:node:links *dag:right*) '(,*dag:left*) )
    (loop (initial glugg (sort (dag:add-link-depths nodes left right)
                               '(lambda (x y) (< (car x) (car y) ) ) )
                   count (length nodes)
                   list  nil
                   node1 nil
                   node2 nil)
          (until (<= count 0) )
          (next list (pop glugg) )
          (next node1 (cadr  list) )
          (next node2 (caddr list) )
          (do (if (== node2 *dag:left*)
                  (:= node2 (dag:get-end *dag:left* *dag:right*) ) )
              (if (== node2 *dag:right*)
                  (:= node2 (dag:get-end *dag:right* *dag:left*) ) )
              (? ( (dag:joined node1 node2)
                   nil)
                 ( (&& (! (cdr (dag:node:links node1) ) )
                       (! (cdr (dag:node:links node2) ) ) )
                   (push (dag:node:links node1) node2)
                   (push (dag:node:links node2) node1)
                   (:= count (+ -1 count) ) ) ) )
          (result (dag:get-sorted-list nodes) ) ) )

(defun dag:get-end (node wrong-way)
    (loop (initial temp nil)
          (while (cdr (dag:node:links node) ) )
          (next temp node)
          (next node (if (== wrong-way (car (dag:node:links node) ) )
                         (cadr (dag:node:links node) )
                         (car  (dag:node:links node) ) ) )
          (next wrong-way temp)
          (result node) ) )

(defun dag:get-sorted-list (nodes)
    (let ( (last (for (node in '(,*dag:left* ,*dag:right* ,,nodes) )
                      (when (! (cdr (dag:node:links node) ) ) )
                      (save node) ) ) )
        (push (dag:node:links (cadr last) ) (car  last) )
        (push (dag:node:links (car  last) ) (cadr last) ) )
    (loop (initial res ()
                   last    *dag:left*
                   current *dag:right*
                   temp    nil)
```

```
        (do (:= temp current)
            (if (== last (car (dag:node:links current) ) )
                (:= current (cadr (dag:node:links current) ) )
                (:= current (car  (dag:node:links current) ) ) )
            (:= last temp) )
        (until (== current *dag:left*) )
        (do (push res current) )
        (result res) ) )

(defun dag:joined (node1 node2)
    (loop (initial res     nil
                   nodes   '(,node1)
                   node    ()
                   seen    () )
          (while nodes)
          (until res)
          (next node (pop nodes) )
          (do (? ( (== node node2)
                   (:= res t) )
                 ( t
                   (push seen node)
                   (for (link in (dag:node:links node) )
                        (when (! (memq link seen) ) )
                        (do (push nodes link) ) ) ) ) )
          (result res) ) )

(defun dag:sort-graph ()
    (:= *dag:graph*  () )
    (:= *dag:edges*  () )
    (:= *dag:widths* () )
    (:= *dag:graph-depth*  0)
    (loop (initial level *dag:front*
                   depth 1)
          (while level)
          (next level (dag:sort-level level) )
          (do (push *dag:graph* level) )
          (next level (for (node in level)
                           (save (dag:node:children node) ) ) )
          (next level (dag:clean-level level) )
          (next depth (+ 1 depth) ) )
    (:= *dag:graph-depth*  (+ -1 (length *dag:graph*) ) )
    (:= *dag:widths* (reverse *dag:widths*) )
    (:= *dag:edges*  (reverse *dag:edges*) )
    (:= *dag:graph*  (reverse *dag:graph*) ) )

(defun dag:clean-level (level)
    (let ( (count 0) )
;;;     (:= level (dag:join-adjacent level) )
        (for (node in (dag:list-nodes level) )
             (do (:= count 0)
                 (for (list in level)
                      (when (memq node list) )
                      (do (:= count (+ 1 count) ) ) )
                 (dag:set-mommy node count)
                 (if (< 1 count)
                     (:= level (dag:remove-extras node
                                                  (// (+ 1 count) 2)
                                                  level) ) ) ) )
        (for (list in level)
             (when list)
             (save list) ) ) )

(defun dag:remove-extras (node count level)
```



```
    (for (list in level)
         (save (if (&& (memq node list)
                       (!= 0 (:= count (+ -1 count) ) ) )
                   (for (n in list)
                        (when (!== node n) )
                        (save n) )
                   list) ) ) )

(defun dag:list-nodes (level)
    (let ( (list () ) )
         (for (elts in level)
              (do (for (node in elts)
                       (when (! (memq node list) ) )
                       (do (push list node) ) ) ) )
         list) )

(defun dag:join-adjacent (level)
    (loop (initial current  ()
                   next     ()
                   adjacent ()
                   done     ()
                   to-do    level)
          (while (cdr to-do) )
          (next current (car to-do) )
          (next next (cadr to-do) )
          (next adjacent (for (node in current)
                              (when (memq node next) )
                              (save node) ) )
          (do (? ( adjacent
                   (push done current)
                   (:= current adjacent)
                   (for (node in adjacent)
                        (do (:= done (dag:remove-extras node -1 done) )
                            (:= to-do (dag:remove-extras node -1 to-do) ) ) ) )
          (next done '(,current ,,done) )
          (next to-do (cdr to-do) )
          (result (reverse '(,,to-do ,,done) ) ) ) )

(defun dag:set-mommy (child number)
    (? ( (= 0 number)
         (:= (dag:node:mommy child) nil) )
       ( (= 1 number)
         (:= (dag:node:mommy child) (car (dag:node:parents child) ) ) )
       ( t
         (loop (initial nodes (car *dag:graph*)
                        node  nil
                        count (// (+ 1 number) 2) )
               (while (< 0 count) )
               (next node (pop nodes) )
               (do (if (memq node (dag:node:parents child) )
                       (:= count (+ -1 count) ) ) )
               (result ;(format t "~&N: ~D M:~D~%"
                       ;          (dag:node:data child)
                       ;          (dag:node:data node) )
                       (:= (dag:node:mommy child) node) ) ) ) ) )
```

```
:clude drawing:dag-declarations)
val-when (compile)
    (build '(drawing:dag) ) )


(defun dag:node-width (node)
    (if (== 'line (dag:node:type node) )
        *dag:line-width*
        *dag:node-width*) )

(defun dag:set-easy-x-coords (edge-node)
    (loop (initial left-limit  20
                   node        edge-node
                   x-coord     0)
          (until (eq node *dag:right*) )
          (next x-coord (dag:get-x-coord node)
                left-limit (+ left-limit (// (dag:node-width node) 2) ) )
          (do (? ( (&& x-coord
                       (< x-coord left-limit) )
                   (dag:move-children node (- left-limit x-coord) )
                   (:= (dag:node:x-coord node) left-limit) )
                 ( x-coord
                   (:= (dag:node:x-coord node) x-coord) )
                 ( t
                   (:= (dag:node:x-coord node) left-limit) ) ) )
          (next left-limit (+ (dag:node:x-coord node)
                              (// (dag:node-width node) 2) ) )
          (next node (dag:node:right node) ) ) )

(defun dag:get-x-coord (node)
    (let ( (total 0)
           (count 0) )
         (for (child in (dag:node:children node) )
              (when (! (cdr (dag:node:parents child) ) ) )
              (do (:= total (+ total (+ (dag:node:x-coord child)
                                        (dag:node:delta-x child) ) ) )
                  (:= count (+ 1 count) ) ) )
         (if (= count 0)
             nil
             (// total count) ) ) )

(defun dag:move-children (parent delta-x)
    (loop (initial node (dag:left-most-child parent) )
          (until (eq node *dag:right*) )
          (do (:= (dag:node:delta-x node)
                  (+ (dag:node:delta-x node)
                     delta-x) ) )
          (next node (dag:node:right node) ) ) )

(defun dag:left-most-child (parent)
    (let ( (node    nil)
           (x-coord 100000) )
         (for (child in (dag:node:children parent) )
              (when (> x-coord (dag:node:x-coord child) ) )
              (do (:= node    child)
                  (:= x-coord (dag:node:x-coord child) ) ) )
         node) )

(defun dag:propogate-moves (edge-node)
    (loop (initial node        edge-node
                   delta-x     0)
          (until (eq node *dag:right*) )
          (next delta-x (max delta-x (dag:mommy-delta-x node) ) )
          (do (:= (dag:node:delta-x node) delta-x) )
          (next node (dag:node:right node) ) ) )

(defun dag:mommy-delta-x (node)
    (if (dag:node:mommy node)
        (dag:node:delta-x (dag:node:mommy node) )
        0) )

(defun dag:do-moves (edge-node)
    (loop (initial node edge-node)
          (until (eq node *dag:right*) )
          (do (:= (dag:node:x-coord node)
                  (+ (dag:node:x-coord node)
                     (dag:node:delta-x node) ) )
              (:= (dag:node:delta-x node) 0) )
          (next node (dag:node:right node) ) ) )

(defun dag:set-x-coords ()
    (dag:set-easy-x-coords (nth-elt *dag:edges* (+ 1 *dag:graph-depth*) ) )
    (loop (decr up from *dag:graph-depth* to 1) (do
        (dag:set-easy-x-coords (nth-elt *dag:edges* up) )

        (loop (incr down from (+ 1 up) to *dag:graph-depth*) (do
            (dag:propogate-moves (nth-elt *dag:edges* (+ 1 down) ) )
            (dag:do-moves        (nth-elt *dag:edges* down) ) )
        (result
            (dag:do-moves (nth-elt *dag:edges* (+ 1 *dag:graph-depth*))))))))



(defun dag:set-y-coords ()
    (loop (for level in *dag:graph*)
          (initial y           0
                   max-delta-x 0
                   delta-y     0)
    (do
                ;*** Find the maximum horizontal distance between any
                ;*** node on this level and its parent.
                ;
        (:= max-delta-x 0)
        (loop (for node in level) (do
            (loop (for parent-node in (dag:node:parents node) ) (do
                (:= max-delta-x
                    (max max-delta-x
                         (abs (- (dag:node:x-coord node)
                                 (dag:node:x-coord parent-node) ) ) ) ) ) ) ) )))

                ;*** Set the separation between this level and the previous
                ;*** to make sure there is at least a minimum % grade
                ;*** on each edge.
                ;
        (:= delta-y
            (max *dag:minimum-level-separation*
                 (fix (times *dag:minimum-edge-grade* max-delta-x) ) ) )

        (:= y (+ y (+ delta-y *dag:box-height*) ) )
        (loop (for node in level) (do
            (:= (dag:node:y-coord node) y) ) )
        () ) ) )
```

```
;=============================================================================
;
; Declarations for the DAG-drawing modules.  This file should be INCLUDEd
; by every module.
;
;=============================================================================

;*****************************************************************************
;***
;*** User-settable parameters that control formatting.  Note that
;*** widths and heights are in arbitrary relative units (x- and y- units
;*** are the same).
;***
;*****************************************************************************

(defvar *dag:paper* 'letter)
        ;
        ;*** One of LETTER, B4, or LEGAL.  This is the size of the paper
        ;*** to use.  B4 is 10 x 14.

(defvar *dag:max-x-panes* 1)
(defvar *dag:max-y-panes* 1)
        ;
        ;*** The number of "panes" (pieces of paper) to use across each
        ;*** dimension.   Setting these to be > 1 causes multi-sheet output.

(defvar *dag:box-height*   46)
(defvar *dag:node-width*  100)
(defvar *dag:box-width*    80)
(defvar *dag:line-width*   25)
        ;
        ;*** *DAG:BOX-HEIGHT*  -- THE HEIGHT OF A BOX.
        ;*** *DAG:NODE-WIDTH*  -- The separation between box centers.
        ;*** *DAG:BOX-WIDTH*   -- The width of a box.
        ;*** *DAG:LINE-WIDTH*  -- The width of space surround a vertical line.

(defvar *dag:minimum-level-separation* 30)
        ;
        ;*** The minimum separation between levels in the DAG (measured from
        ;*** from the bottom of the upper box to the top of the lower box).

(defvar *dag:minimum-edge-grade* 0.15)
        ;
        ;*** The minimum y/x ratio for each diagonal edge.  The separation
        ;*** between levels will be increased until the grade on each
        ;*** edge is at least this much.

(defvar *dag:ideal-text-lines-per-node* 4)
        ;
        ;*** The number of text lines desired for each box.  The actual number
        ;*** of lines will be larger or smaller depending on the actual
        ;*** point sizes of available fonts.

(defvar *dag:text-x-margin*             3)
        ;
        ;*** The margin between the left and right edges of the box and text
        ;*** within the box.

(defvar *dag:remove-non-critical-nodes&edges?* () )
        ;
        ;*** If true, nodes and edges that aren't on a critical path are
        ;*** removed before printing.
```

1

```
(defvar *dag:shade-critical-nodes?* t)
        ;
        ;*** If true, nodes on critical paths are shaded.

(defvar *dag:critical-threshold* 0)
        ;
        ;*** Defines the length of what are considered critical paths.
        ;*** A path is "critical" if its length is >= the depth of the
        ;*** DAG minus the threshold.


(defvar *dag:make-slides?*  () )
        ;
        ;*** If T then no shading is done;  THICK lines are replaced with
        ;*** normal lines, and SHADED lines are replaced with THICK
        ;*** lines (what a hack).

;*****************************************************************************
;***
;*** These variables are NOT user-settable.
;***
;*****************************************************************************

(defvar *dag:flag*                  0)
(defvar *dag:front*                 () )
(defvar *dag:graph*                 () )
(defvar *dag:graph-input*           () )
(defvar *dag:edges*                 () )
(defvar *dag:graph-depth*           0)
(defvar *dag:depth*                 0)
(defvar *dag:height*                0)
(defvar *dag:widths*                () )
(defvar *dag:lines-per-node*         4)
(defvar *dag:text-height*           13)
(defvar *dag:text-height-points*     0)
(defvar *dag:max-string*            10)
(defvar *dag:flag*                  () )
(defvar *dag:left*                  () )
(defvar *dag:right*                 () )
(defvar *dag:max-x*                 0)
(defvar *dag:max-y*                 0)
```

2

```
:nclude drawing:dag-declarations)
:val-when (compile)
   (build '(drawing:dag) ) )

(declare (special                ;*** Imported from IMPRESS-DRAW.
   *id.paper-x-y-ratio*
   *id.cmasc-point-sizes*
   ) )

(defun dag:impress-graph (file-name)
   (unwind-protect
      (let ( (x-pane   0)
             (y-pane   0)
             (x-margin 0)
             (y-margin 0) )
         (id.start-job file-name *dag:paper*)

                ;*** Figure out the width and height of each pane.
                ;
         (dag:set-maximum-coords)
         (:= x-pane (ceiling-divide *dag:max-x* *dag:max-x-panes*) )
         (:= y-pane (ceiling-divide *dag:max-y* *dag:max-y-panes*) )
         (if (> (// (flonum *dag:max-x*)
                    (* (flonum y-pane) *id.paper-x-y-ratio*) )
                (flonum *dag:max-x-panes*) )
         (then
            (:= y-pane (fix (+ (// (flonum x-pane) *id.paper-x-y-ratio*)
                               0.999999) ) ) )

         (else
            (:= x-pane (fix (+ (* (flonum y-pane) *id.paper-x-y-ratio*)
                               0.999999) ) ) ) )

         (msg 0 "Paper: " *dag:paper* t)
         (msg   (ceiling-divide *dag:max-x* x-pane) " x-panes by "
                (ceiling-divide *dag:max-y* y-pane) " y-panes." t)

         (id.set-coordinates 0 0 (+ -1 x-pane) (+ -1 y-pane) )
         (:= x-margin (fix (id.delta-x:delta-lx 50) ) )
         (:= y-margin (fix (id.delta-y:delta-ly 50) ) )

                ;*** Set the text size information
                ;
         (dag:set-text-info)
         (id.define-font-size *dag:text-height-points*)

                ;*** Make repeated passes over the picture, shifting
                ;*** the view pane from top to bottom, left to right,
                ;*** redrawing the picture for each pane.
                ;
         (loop (incr x from 0 by x-pane)
               (while (< x *dag:max-x*) )
         (do
            (loop (incr y from 0 by y-pane)
                  (while (< y *dag:max-y*) )
            (do
               (id.begin-page)
               (id.set-coordinates (- x x-margin) (- y y-margin)
                                   (+ x (+ x-pane x-margin) )
                                   (+ y (+ y-pane y-margin) ) )

               (for (list in *dag:graph*) (do
                  (for (node in list) (do
```

1

```
                        (dag:plot-node node) ) ) ) )

               (dag:cut-marks x y
                              (+ x x-pane) (+ y y-pane) )
               (id.end-page) ) ) ) )
         () )

      (id.end-job) ) )


(defun dag:set-text-info ()
   (let*( (point-size 0)
          (node-text-height (times .9 *dag:box-height*) )
          (ideal-point-size
             (id.delta-ly:points
                (// node-text-height
                   (flonum *dag:ideal-text-lines-per-node*) ) ) ) )

             ;*** First, find the largest point size we can use for CMASC
             ;
      (loop (for size in *id.cmasc-point-sizes*) (do
         (if (<= (flonum size) ideal-point-size) (then
             (:= point-size size)
             (return () ) ) ) ) )
      (result
         (:= point-size (last-elt *id.cmasc-point-sizes*) ) ) )

             ;*** Set the height of each line and the actual number
             ;*** of lines per node accordingly
             ;
      (:= *dag:text-height-points* point-size)
      (:= *dag:text-height* (round (id.points:delta-ly point-size) ) )
      (:= *dag:lines-per-node*
         (fix (// node-text-height (id.points:delta-ly point-size) ) ) )

             ;*** Set the number of chars allowed per line.
             ;***
             ;
      (:= *dag:max-string* (fix
         (id.delta-lx:chars (- *dag:box-width* (* 2 *dag:text-x-margin*))
                            point-size) ) )
      () ) )


(defun dag:set-maximum-coords ()
   (:= *dag:max-x* 0)
   (:= *dag:max-y* 0)

   (for (ls in *dag:graph*)
        (do (for (node in ls)
                 (do (if (<  *dag:max-x* (dag:node:x-coord node) )
                         (:= *dag:max-x* (dag:node:x-coord node) ) ) ) ) ) )
   (:= *dag:max-x* (+ *dag:max-x* *dag:node-width*) )
   (:= *dag:max-y* (+ (dag:node:y-coord (car (last-elt *dag:graph*) ) )
                      *dag:box-height*) )
   (:= *dag:max-x* (+ *dag:max-x* 20) )
   (:= *dag:max-y* (+ *dag:max-y* 20) ) )


(defun dag:cut-marks ( x-min y-min x-max y-max )
   (let ( (dx    (fix (id.delta-x:delta-lx 50) ) )
          (dy    (fix (id.delta-y:delta-ly 50) ) ) )
      (dag:cut-mark x-min y-min dx dy)
```

2

PS:<C.S.BULLDOG.DRAWING>DAG-IMPRESS.LSP.10

```
        (dag:cut-mark x-max y-min dx dy)
        (dag:cut-mark x-min y-max dx dy)
        (dag:cut-mark x-max y-max dx dy)
        () ) )

(defun dag:cut-mark ( x y dx dy )
    (id.draw-line (- x dx) y
                  (+ x dx) y
                  'normal)
    (id.draw-line x (- y dy)
                  x (+ y dy)
                  'normal)
    () )


(defun dag:plot-node ( node )
    (let ( (x (dag:node:x-coord node) )
           (y (dag:node:y-coord node) ) )

        (if (== 'line (dag:node:type node) ) (then
            (id.draw-line x y x (+ y *dag:box-height*)
                          (dag:node:style node) ) )

        (else
            (dag:plot-node-text x y (dag:node:label node) )
            (id.draw-box (- x (// *dag:box-width* 2) )
                         y
                         *dag:box-width*
                         *dag:box-height*
                         (dag:node:style node) ) ) )

        (for (child in (dag:node:children node) ) (do
            (id.draw-line
                x
                (+ y *dag:box-height*)
                (dag:node:x-coord child)
                (dag:node:y-coord child)
                (dag:node:child-edge-style node child) ) ) ) ) )


(defun dag:plot-node-text (x y text-lines)

(let ( (lines () ) )

    (loop (for line in text-lines) (do
        (loop (initial line-list (aexplodec line) )
        (while line-list)
        (do
            (push lines
                  (firstn line-list *dag:max-string*) ) )
        (next line-list (nth line-list (+ 1 *dag:max-string*) ) ) ) )
    (result
        (:= lines (dreverse lines) ) ) )

    (loop (incr i from 1 to *dag:lines-per-node*)
          (incr line-y from (+ y *dag:text-height*) by *dag:text-height*)
          (for line in lines)
          (initial line-x (+ (- x (// *dag:box-width* 2) )
                             *dag:text-x-margin*) )
    (do
        (id.draw-text line line-x line-y *dag:text-height-points*) ) )
    () ) )
```



```
(defun ceiling-divide ( x y )
    (// (+ x (+ -1 y) )
        y) )
```

```
;***
;*** This structure defines the communication between the DAG-drawing
;*** functions and the compiler.  The compiler writes a file with a list
;*** of these records, which is read in by the drawing functions.  The
;*** compiler is too big to run with the DAG-drawing stuff.
;***

(def-struct dag:input-node
    name                    ;*** A unigue name (or number) given to this
                            ;*** node.

    child-edge-styles       ;*** A list of pairs (CHILD-NAME EDGE-STYLE)
                            ;*** describing edges from this node to child
                            ;*** nodes.

    label                   ;*** The text label to be printed for this node.

    style                   ;*** The style (shaded or not) of the box.
    )
```

```
(include drawing:dag-declarations)
(eval-when (compile)
    (build '(drawing:dag) ) )

(def-struct dag:front
    nodes
    depth
    flag
    )

(defun dag:new-flag ()
    (:= *dag:flag* (+ 1 *dag:flag*) ) )

(defun dag:add-link-depths (nodes left right)
    (loop (initial res   ()
                   node1 () )
          (while nodes)
          (next node1 (pop nodes) )
          (next res '( (,(dag:link-depth node1 left)  ,node1 ,*dag:left*)
                       (,(dag:link-depth node1 right) ,node1 ,*dag:right*)
                       ,,(for (node2 in nodes)
                              (save '(,(dag:link-depth node1 '(,node2) )
                                        ,node1
                                        ,node2) ) )
                       ,,res) )
          (result res) ) )

(defun dag:link-depth (node nodes)
    (let ( (link (dag:find-link '(,node) nodes) ) )
         (if link
             (dag:node:depth link)
             1000000) ) )

(defun dag:find-link (nodes1 nodes2)
    (let ( (flag1 (dag:new-flag) )
           (flag2 (dag:new-flag) ) )
         (loop (initial link nil
                        front1 (dag:new-front nodes1 flag1)
                        front2 (dag:new-front nodes2 flag2) )
               (while (&& (dag:front:nodes front1)
                          (dag:front:nodes front2) ) )
               (next link (dag:next-level front1 flag2) )
               (until link)
               (next link (dag:next-level front2 flag1) )
               (until link)
               (result link) ) ) )

(defun dag:new-front (nodes flag)
    (dag:front:new nodes nodes
                   depth (if nodes
                             (dag:node:depth (car nodes) )
                             1000000)
                   flag  flag) )

(defun dag:next-level (front search-flag)
    (let ( (link nil) )
         (:= (dag:front:nodes front)
             (for (node in (dag:front:nodes front) )
                  (splice (cond ( (= (dag:front:depth front)
                                     (dag:node:depth node) )
                                  (if (= (dag:node:flag node) search-flag)
                                      (:= link node)
                                      (:= (dag:node:flag node)
```

1

```
                                          (dag:front:flag front) ) )
                                  (dag:copy (dag:node:children node) ) )
                                ( t
                                  '(,node) ) ) ) ) )
        (:= (dag:front:depth front) (+ 1 (dag:front:depth front) ) )
        link) )
```

2

PS:<C.S.BULLDOG.DRAWING>DAG-LINK.LSP.3

```
(include drawing:dag-declarations)
(eval-when (compile)
    (build '(
        utilities:sharp-sharp
        drawing:dag-input-node
        drawing:dag) ) )

(defun dag:read-graph ( filename )
    (:= *dag:graph*       () )
    (:= *dag:front*       () )
    (:= *dag:graph-input* () )

    (iota ( (file filename '(in old) ) ) (within file
        (loop (for command in (read) ) (do
            (eval command) ) )
        (:= *dag:graph-input* (read) ) ) )

    (loop (for input-node in *dag:graph-input*) (do
        (push *dag:graph*
              (dag:node:new name
                                (dag:input-node:name  input-node)
                            style
                                (dag:input-node:style input-node)
                            label
                                (dag:input-node:label input-node)
                            children
                                (for ( (child style) in
                                       (dag:input-node:child-edge-styles
                                           input-node) )
                                    (save child) )
                            child-edge-styles
                                 (dag:input-node:child-edge-styles
                                    input-node) ) ) ) )

    (dag:clean-graph)
    (dag:count-parents)
    (dag:set-depths)
    (dag:set-heights)
    (if *dag:shade-critical-nodes?* (then
        (dag:shade-critical-nodes) ) )
    (if *dag:remove-non-critical-nodes&edges?* (then
        (dag:remove-non-critical-nodes&edges) ) )
    (dag:set-depths)
    (dag:set-heights)
    (dag:set-front)
    (dag:add-lines)
    (if *dag:make-slides?*
        (dag:make-slides) )
    () )


(defun dag:clean-graph ()
    (loop (for node in *dag:graph*) (do
        (:= (dag:node:children node)
            (for (name in (dag:node:children node) ) (save
                (dag:name:node name) ) ) ) ) )

    (loop (for node in *dag:graph*) (do
        (:= (dag:node:child-edge-styles node)
            (for ( (name style) in (dag:node:child-edge-styles node) )
            (save
                `(,(dag:name:node name)
                  ,style) ) ) ) ) )
```

1

```
    () )

(defun dag:count-parents ()
    (for (node in *dag:graph*) (do
        (:= (dag:node:flag node) 0) ) )

    (for (node in *dag:graph*) (do
        (for (child in (dag:node:children node) ) (do
            (push (dag:node:parents child) node) ) ) ) ) )


(defun dag:set-front ()
    (:= *dag:front*
        (list (for (node in *dag:graph*)
                   (when (! (dag:node:parents node) ) )
                   (save node) ) ) ) )

(defun dag:set-heights ()
    (:= *dag:height* -1)
    (loop (for node in *dag:graph*) (do
        (:= (dag:node:height node) -1) ) )

    (loop (for node in *dag:graph*)
          (when (! (dag:node:parents node) ) )
    (do
        (:= *dag:height*
            (max *dag:height* (dag:node:set-height node) ) ) ) )
    () )

(defun dag:node:set-height ( node )
    (if (!= -1 (dag:node:height node) ) (then
        (dag:node:height node) )
    (else
        (loop (for child in (dag:node:children node) )
              (initial height -1)
        (do
            (:= height (max height (dag:node:set-height child) ) ) )
        (result
            (:= (dag:node:height node) (+ 1 height) ) ) ) ) ) )


(defun dag:set-depths ()
    (:= *dag:depth* -1)
    (loop (for node in *dag:graph*) (do
        (:= (dag:node:depth node) -1) ) )

    (loop (for node in *dag:graph*)
          (when (! (dag:node:children node) ) )
    (do
        (:= *dag:depth*
            (max *dag:depth* (dag:node:set-depth node) ) ) ) )
    () )

(defun dag:node:set-depth ( node )
    (if (!= -1 (dag:node:depth node) ) (then
        (dag:node:depth node) )
    (else
        (loop (for parent in (dag:node:parents node) )
              (initial depth -1)
        (do
            (:= depth (max depth (dag:node:set-depth parent) ) ) )
        (result
            (:= (dag:node:depth node) (+ 1 depth) ) ) ) ) ) )
```

2

PS:<C.S.BULLDOG.DRAWING>DAG-UTIL.LSP.7

```
(defun dag:shade-critical-nodes ()
    (let ( (threshold (if *dag:remove-non-critical-nodes&edges?*
                          *dag:depth*
                          (- *dag:depth* *dag:critical-threshold*) ) ) )
        (loop (for node in *dag:graph*) (do
            (if (>= (+ (dag:node:depth  node)
                       (dag:node:height node) )
                    threshold)
                (then
                    (:= (dag:node:style node) 'shaded) ) ) ) ) ) )


(defun dag:remove-non-critical-nodes&edges ()
    (:= *dag:graph*
        (loop (for node in *dag:graph*)
              (initial new-graph () )
        (do
            (if (>= (+ (dag:node:depth node) (dag:node:height node) )
                    (- *dag:depth* *dag:critical-threshold*) )
                (then
                    (push new-graph node) )
                (else
                    (dag:node:delete node) ) ) )
        (result new-graph) ) )

    (loop (for node in *dag:graph*) (do
        (loop (for child in (dag:node:children node) ) (do
            (if (> (- (dag:node:depth child)
                      (dag:node:depth node) )
                   (+ 2 *dag:critical-threshold*) )
                (then
                    (:= (dag:node:children node)
                        (top-level-removeq child (dag:node:children node) ) )
                    (:= (dag:node:parents child)
                        (top-level-removeq node (dag:node:parents child) )))))))
    () )


(defun dag:node:delete ( node )
    (loop (for child in (dag:node:children node) ) (do
        (:= (dag:node:parents child)
            (top-level-removeq node (dag:node:parents child) ) ) ) )

    (loop (for parent in (dag:node:parents node) ) (do
        (:= (dag:node:children parent)
            (top-level-removeq node (dag:node:children parent) ) ) ) )
    () )



(defun dag:add-lines ()
    (let ( (lines
             (for (node in *dag:graph*) (splice
                (for (child in (dag:node:children          node) )
                     (when (!= (+ 1 (dag:node:depth node) )
                               (dag:node:depth child) ) )
                (splice
                    (dag:make-line node child
                        (dag:node:child-edge-style node child))))))))
```

3

```
        (nconc *dag:graph* lines) ) )

(defun dag:make-line ( node child edge-style )
    (loop (initial last  node
                   line  nil
                   lines () )
          (until (>= (+ 1 (dag:node:depth last) )
                     (dag:node:depth child) ) )
          (next line (dag:node:new type  'line
                                  style edge-style
                                  depth (+ 1 (dag:node:depth last) ) ) )
    (do
        (push (dag:node:children last) line)
        (push (dag:node:child-edge-styles last)
              '(,line ,edge-style) )
        (push (dag:node:parents line) last)
        (push lines line) )
    (next last line)
    (result
        (push (dag:node:children last) child)
        (push (dag:node:child-edge-styles last)
              '(,child ,edge-style) )
        (:= (dag:node:children node)
            (for (n in (dag:node:children node) )
                 (when (!== n child) )
                 (save n) ) )
        (push (dag:node:parents child) last)
        (:= (dag:node:parents child)
            (for (n in (dag:node:parents child) )
                 (when (!== n node) )
                 (save n) ) )
        lines) ) )

(defun dag:copy (list)
    (for (elt in list)
         (save elt) ) )

(defun dag:name:node ( name )
    (loop (for node in *dag:graph*) (do
        (? ( (consp node)
             (if-let ( (result
                         (loop (for sub-node in node) (do
                            (if (= name (dag:node:name sub-node) ) (then
                                (return sub-node) ) ) ) ) ) )
                 (return result) ) )
           ( (= name (dag:node:name node) )
             (return node) ) ) ) ) )

(defun dag:node:child-edge-style ( node child )
    (cadr (assoc child (dag:node:child-edge-styles node) ) ) )

(def-sharp-sharp n
    (dag:name:node (read) ) )


(defun dag:make-slides ()
    (loop (for node in *dag:graph*) (do
        (if (== 'line (dag:node:type node) ) (then
            (:= (dag:node:style node)
                (caseq (dag:node:style node)
                    (shaded 'thick)
                    (thick  'normal)
                    (t      'normal) ) ) )
```

4

PS:<C.S.BULLDOG.DRAWING>DAG-UTIL.LSP.7

```
        (else
            (:= (dag:node:style node) () ) ) )

        (:= (dag:node:child-edge-styles node)
            (for ( (child style) in (dag:node:child-edge-styles node) ) (save
                '(,child
                  ,(caseq style
                       (shaded 'thick)
                       (thick  'normal)
                       (t      'normal) ) ) ) ) ) ) )
```

```
;============================================================================
;
; IMPRESS
;
; This module provides an interface to the ImPress language for the Imagen
; printer.  There is generally a one-to-one mapping between ImPress
; commands and functions; for example, to invoke the SET-PEN operation
; with the pen-width argument:
;
;     (im.set-pen 15)
;
; Numeric arguments are assumed to be fixnums, except for x-y coordinates,
; which may be fixnums or floats (which are rounded to fixnums).
;
; The ImPress file is written using pseudo-hex, where each ASCII character
; represents a 4-bit nibble offset by "A".  Bytes are written as two
; nibbles in the inconsistent order high byte, low byte.
;
; Only the non-obvious functions are described here:
;
; (IM.START-JOB FILE-NAME PAPER)
;     This should be called before any other function.  It begins writing
;     ImPress commands to file FILE-NAME, selecting paper-type PAPER,
;     which should be of type LETTER, B4, or LEGAL.
;
; (IM.END-JOB)
;     This should be the last function called.  It closes the output file.
;
; (IM.CREATE-PATH COORDINATES)
;     Invokes the CREATE-PATH ImPress command.  COORDINATES is a list
;     of pairs (X Y), each pair describing a point on the path.
;
; (IM.GLYPH:NEW WIDTH HEIGHT)
;     Creates a new glyph structure.  A glyph is represented as a list
;     of rows, each row a list of fixnums, each representing an 8-bit
;     byte.
;
; (IM.GLYPH:SET-PIXEL GLYPH X Y VALUE)
;     Sets the X-Yth pixel of a glyph to be VALUE (1 or 0).
;
; (IM.GLYPH:LOGAND G1 G2)
; (IM.GLYPH:LOGIOR G1 G2)
; (IM.GLYPH:LOGNOT G1)
;     Non-destructive logical operations on two glyphs.
;
; (IM.GLYPH:READ WIDTH HEIGHT)
;     Reads a WIDTH x HEIGHT glyph from the current input stream; an "@"
;     represents a 1, "."  a 0.  See IMPRESS-TEXTURES for examples of
;     this.
;
; (IM.TEXTURE:READ)
;     Reads a texture glyph;  equivalent to (IM.GLYPH:READ 32 32).
;
; (IM.GLYPH:PRINT G)
;     Pretty prints a glyph to the current output; "@" is used for 1,
;     "."  for 0.
;
; (IM.DUMP-FILE &OPTIONAL (FILENAME "DAG.IMPA"))
;     Prints out a symbolic dump of FILENAME, which is assumed to contain
;     ImPress commands.
;
;============================================================================
```

1

```
(defvar *im.rasters-per-inch* 240)

(defvar *im.file* () )

(defun im.write-8 ( byte )
    (without *im.file*
        (tyo (+ (// byte 16) #/A) )
        (tyo (+ (\  byte 16) #/A) )
        () ) )

(defun im.write-string ( string )
    (loop (for char in (aexplodec string) ) (do
        (im.write-8 char) ) )
    () )

(defun im.write-string-zero ( string )
    (im.write-string string)
    (im.write-8 0)
    () )

(defun im.write-8s ( byte-list )
    (loop (for byte in byte-list) (do
        (im.write-8 byte) ) )
    () )

(defun im.write-16 ( word )
    (let ( (high (ldb32 word 8 8) )
           (low  (ldb32 word 0 8) ) )
        (im.write-8 high)
        (im.write-8 low)
        () ) )

(defun im.read-8 ()
    (within *im.file*
        (let*( (high (tyi) )
               (low  (tyi) ) )
            (+ (* 16 (- high #/A) )
               (- low #/A) ) ) ) )

(defun im.read-16 ()
    (let*( (high   (im.read-8) )
           (low    (im.read-8) )
           (result 0) )
        (if (>= high 128) (then
            (:= result -1) ) )
        (:= result (dpb32 result 8 8 high) )
        (:= result (dpb32 result 0 8 low) )
        result) )

(defun im.read-string ()
    (loop (initial char-list ()
                   char      0)
          (while (!= 0 (:= char (im.read-8) ) ) )
    (do
        (push char-list char) )
    (result
        (packc (dreverse char-list) ) ) ) )

(defun im.start-job ( file-name paper )
    (:= *im.file* (openf file-name '(out) ) )
```

2

PS:<C.S.BULLDOG.DRAWING>IMPRESS.LSP.21

```
    (im.write-string
        (string-msg "@document(owner Ellis, name /"" (c file-name)
                    "/",jobheader on,language impress,paper "
                    (c paper) ",pagereversal off)" ) )
    () )

(defun im.end-job ()
    (im.write-8 255)                    ;*** End of *im.file*
    (:= *im.file* (closef *im.file*) )
    () )


(defun im.begin-page ()
    (im.write-8 213)
    () )

(defun im.end-page ()
    (im.write-8 219)
    () )


(defun im.define-font ( font-name point-size font-id )
    (im.write-8 250)
    (im.write-8 font-id)
    (im.write-8 point-size)
    (im.write-string-zero font-name)
    () )

(defun im.set-family ( font-id )
    (im.write-8 207)
    (im.write-8 font-id)
    () )


(defun im.create-family-table ( family mapnumber&fontname-list )
    (im.write-8 221)
    (im.write-8 family)
    (im.write-8 (length mapnumber&fontname-list) )
    (loop (for (mapnumber fontname) in mapnumber&fontname-list) (do
        (im.write-8 mapnumber)
        (im.write-string-zero fontname) ) )
    () )

(defun im.create-map ( mapnumber memberbase&symbolbase&count-list )
    (im.write-8 222)
    (im.write-8 mapnumber)
    (im.write-8 (length memberbase&symbolbase&count-list) )
    (loop (for (memberbase symbolbase count) in
               memberbase&symbolbase&count-list)
    (do
        (im.write-8  memberbase)
        (im.write-16 symbolbase)
        (im.write-8  count) ) )
    () )


(defun im.push ()
    (im.write-8 211)
    () )

(defun im.pop ()
    (im.write-8 212)
    () )
```



```
(defun im.set-xpos ( xpos )
    (im.write-8 195)
    (im.write-16 (* 2 (round xpos) ) )
    () )

(defun im.set-ypos ( ypos )
    (im.write-8 196)
    (im.write-16 (* 2 (round ypos) ) )
    () )


(defun im.set-xpos-relative ( xpos )
    (im.write-8 195)
    (im.write-16 (+ 1 (* 2 (round xpos) ) ) )
    () )

(defun im.set-ypos-relative ( ypos )
    (im.write-8 196)
    (im.write-16 (+ 1 (* 2 (round ypos) ) ) )
    () )


(defun im.set-pen ( type size )
    (im.write-8 232)
    (im.write-8 size)
    () )


(defun im.set-texture ( family member )
    (im.write-8 231)
    (im.write-16 (+ (lsh family 7) member) )
    () )


(defun im.create-path ( coordinates )
    (im.write-8  230)
    (im.write-16 (length coordinates) )
    (loop (for point in coordinates) (do
        (im.write-16 (round (car  point) ) )
        (im.write-16 (round (cadr point) ) ) ) )
    () )

(defun im.draw-path ( operation )
    (im.write-8 234)
    (im.write-8 operation)
    () )

(defun im.fill-path ( operation )
    (im.write-8 233)
    (im.write-8 operation)
    () )


(defun im.small-glyph ( orientation family member advance width
                        left-offset height top-offset glyph )
    (im.write-8 198)
    (im.write-16 (+ (lsh orientation 14) (+
                    (lsh family      7)
                    member) ) )
    (im.write-8 advance)
    (im.write-8 width)
    (im.write-8 left-offset)
```

```
    (im.write-8 height)
    (im.write-8 top-offset)
    (loop (for row in glyph) (do
        (loop (for byte in row) (do
            (im.write-8 byte) ) ) ) )
    () )


(defun im.glyph:new ( width height )
    (loop (incr i from 1 to height)
          (initial row-list () )
    (do
        (loop (incr i from 1 to (// (+ width 7) 8) )
              (initial row () )
        (do
            (push row 0) )
        (result
            (push row-list row) ) ) )
    (result row-list) ) )


(defun im.glyph:set-pixel ( glyph x y value )
    (let ( (byte-cdr (nth (nth-elt glyph (+ 1 y) )
                          (// x 8) ) ) )
        (if (= 1 value) (then
            (:= (car byte-cdr)
                (logior (car byte-cdr)
                        (lsh 128 (- (\ x 8) ) ) ) ) )
        (else
            (:= (car byte-cdr)
                (logand (car byte-cdr)
                        (lognot (lsh 128 (- (\ x 8) ) ) ) ) ) ) )
        glyph) )


(defun im.glyph:logand ( g1 g2 )
    (for (row1 in g1)
         (row2 in g2)
    (save
        (for (byte1 in row1)
             (byte2 in row2)
        (save
            (logand byte1 byte2) ) ) ) ) )

(defun im.glyph:logior ( g1 g2 )
    (for (row1 in g1)
         (row2 in g2)
    (save
        (for (byte1 in row1)
             (byte2 in row2)
        (save
            (logior byte1 byte2) ) ) ) ) )

(defun im.glyph:lognot ( g1 )
    (for (row1 in g1) (save
        (for (byte1 in row1) (save
            (lognot byte1) ) ) ) ) )


(defun im.texture:read ()
    (im.glyph:read 32 32) )
```

5

```
(defun im.glyph:read ( width height )
    (loop (incr y from 0 to (+ -1 height) )
          (initial row ()
                   g   () )
    (do
        (:= row () )
        (loop (incr x from 0 to (+ -1 width) )
              (initial byte 0)
        (do
            (loop (initial c 0)
            (while (! (memq (:= c (tyi) ) '(#/@ #/.) ) ) )
            (result
                (:= byte (dpb32 byte (- 7 (\ x 8) )
                                1
                                (if (= c #/@) 1 0) ) ) ) )

            (if (|| (= 7 (\ x 8) )
                    (= x (+ -1 width) ) )
            (then
                (push row byte) ) ) )

        (result
            (push g (dreverse row) ) ) ) )
    (result (dreverse g) ) ) )


(defun im.glyph:print ( g )
    (loop (for row in g) (do
        (msg 0)
        (loop (for byte in row) (do
            (loop (decr i from 7 to 0) (do
                (msg (c (if (= 1 (ldb32 byte 1 i) ) "@" ".") ) ))))))))


(defun im.dump-file ( &optional (filename "DAG.IMPA") )
    (iota ( (*im.file* filename '(in old) ) ) (within *im.file*

        (msg 0)
        (loop (initial char 0.)
              (next char (im.read-8) )
        (do
            (tyo char) )
        (while (!= #/) char) ) )

        (loop (initial char 0)
              (next char (im.read-8) )
        (do
            (caseq char
                (213 (msg 0 "Beginning of page" t) )
                (219 (msg 0 "End of page" t) )
                (211 (msg 0 "Push" t) )
                (212 (msg 0 "Pop" t) )
                (255
                    (msg 0 "End of job" t)
                    (return () ) )
                ( (195 196)
                    (let ( (pos (im.read-16) ) )
                        (msg 0
                            (c (if (= 195 char) "Set-xpos" "Set-ypos") ) ": "
                            (c (if (== 1 (\ pos 2) ) "+" "") )
                            (// pos 2)
                            t) ) )
```

6

```
(196 (msg 0 "Set-ypos: " (// (im.read-16) 2) t) )
(221
    (let ( (family (im.read-8) )
           (size   (im.read-8) ) )
        (msg 0 "Create family table: family:" family
               " size:" size t)
        (loop (incr i from 1 to size) (do
            (msg "    map:" (im.read-8)
                 " name:" (im.read-string) t) ) ) ) )
(222
    (let ( (number (im.read-8) )
           (size   (im.read-8) ) )
        (msg 0 "Create map: number:" number
               " size:" size t)
        (loop (incr i from 1 to size) (do
            (msg
             "    memberbase:" (im.read-8)
             " symbolbase:"    (im.read-16)
             " count:"         (im.read-8) t) ) ) ) )
(230
    (let ( (size (im.read-16) ) )
        (msg 0 "Create path: " size "  ")
        (loop (incr i from 1 to size) (do
            (msg (im.read-16) ":" (im.read-16) "  ") ) )
        (msg t) ) )
(231
    (let ( (f-m (im.read-16) ) )
        (msg 0 "Set texture: "
               (// f-m 128) (\ f-m 128) t) ) )
(232
    (let ( (type-size (im.read-8) ) )
        (msg 0 "Set pen: "
               (//  type-size 32)
               (\  type-size 32)
               t) ) )
(233 (msg 0 "Fill path: " (im.read-8) t) )
(234 (msg 0 "Draw path: " (im.read-8) t) )
(250 (msg 0 "Define font: id:" (im.read-8)
           " size:" (im.read-8)
           " name:" (im.read-string) t) )
(207 (msg 0 "Set family: " (im.read-8) t) )
(198
    (let ( (o-f-m       (im.read-16) )
           (advance     (im.read-8) )
           (width       (im.read-8) )
           (left-offset (im.read-8) )
           (height      (im.read-8) )
           (top-offset  (im.read-8) ) )
        (msg  0
            "Small glyph: orientation:" (ldb32 o-f-m 14 2)
            " family:"                  (ldb32 o-f-m  7 7)
            " member:"                  (ldb32 o-f-m  0 7)
            t "    advance:"            advance
            " width:"                   width
            " left-offset:"             left-offset
            " height:"                  height
            " top-offset:"              top-offset
            t)
        (loop (incr i from 1 to
                    (* (// (+ width 7) 8) height) )
        (do
            (let ( (*base 8.) )
                (msg (im.read-8) " ") ) ) ) ) )
```



```
            (t
                (if (&& (>= char 0) (<= char 127) ) (then
                    (tyo char) )
                (else
                    (error (list char
                                 "Unexpected byte in input.")))))))
        () ) ) )


(defun im.dump-resident-font ( name symbol-ranges )
    (unwind-protect
        (let ()
            (im.start-job "DAG.IMPA" 'letter)

            (im.create-family-table 1 '( (0 "cmasc10") ) )
            (im.create-family-table 2 '( (0 ,name) ) )

            (im.begin-page)
            (loop (for (begin-symbol end-symbol) in symbol-ranges)
                  (incr i from 0)
            (do
                (loop (incr char from begin-symbol to end-symbol)
                      (initial  row 0 col 0)
                (do
                    (:= col (\ i 8) )
                    (:= row (\ (// i 8) 16) )
                    (im.set-xpos (+ 100 (* col 200) ) )
                    (im.set-ypos (+ 100 (* row 150) ) )
                    (im.set-family 1)
                    (im.write-string (string-msg char) )
                    (if (&& (>  char #/ )
                            (!= char 127) )
                    (then
                        (im.set-xpos-relative 25)
                        (im.write-8 char) ) )
                    (im.set-xpos (+ 100 (* col 200) ) )
                    (im.set-ypos (+ 100 (+ 80 (* row 150) ) ) )
                    (im.set-family 2)
                    (im.write-8 char)

                    (:= i (+ i 1) ) ) ) ) )

            (im.end-page) )
        (im.end-job) )
    (im.dump-file "DAG.IMPA") )
```

```
;;;=============================================================================
;;;
;;; IMPRESS-DRAW
;;;
;;; This module provides functions for drawing simple text, lines, and boxes
;;; on the Imagen, doing efficient clipping (which doesn't currently work
;;; on on the Imagen).  It uses the IMPRESS module to generate a file of
;;; ImPress language commands, and the IMPRESS-TEXTURES module which defines
;;; the textures used here.  The resident Imagen fonts CMASC1 are used for
;;; text; only one font size may be used for any particular job.  X-Y
;;; coordinates may be either fixnums or flonums; all other numeric arguments
;;; should be fixnums.
;;;
;;; (ID.START-JOB FILE-NAME PAPER)
;;;    Starts writing commands to a new file FILE-NAME. PAPER is the type
;;;    of paper to use, and should be one of LETTER, LEGAL, or B4.  This
;;;    function should be called first.
;;; (ID.END-JOB)
;;;    Closes the output file.   This function should be called last.
;;; (ID.BEGIN-PAGE)
;;; (ID.END-PAGE)
;;;    These start and end pages.
;;;
;;; *ID.CMASC-POINT-SIZES*
;;;    List of available font sizes (in points) for text.
;;;
;;; *ID.CMTT-CHARS-PER-EM*
;;;    Fractional number of characters per em space.
;;;
;;; (ID.DEFINE-FONT-SIZE FONT-SIZE)
;;;    Selects FONT-SIZE as the size for text.  This should be called exactly
;;;    once right after START-JOB.
;;;
;;; (ID.SET-COORDINATES LX-MIN LY-MIN X-MAX Y-MAX)
;;;      Sets up the mapping from the client's "logical" coordinates to the actua
;;;    paper coordinates.  Like the Imagen, X coordiantes go left to right,
;;;    Y coordinates top to bottom.
;;;
;;; (ID.LX:X LX)
;;; (ID.LY:Y LY)
;;;    Converts from logical coordinates to paper coordinates.
;;;
;;; (ID.DELTA-LX:DELTA-X LX)
;;; (ID.DELTA-LY:DELTA-Y LY)
;;;    Converts a distance in logical coordinates to a distance
;;;    in paper coordinates.
;;;
;;; (ID.DELTA-X:DELTA-LX X)
;;; (ID.DELTA-Y:DELTA-LY Y)
;;;    Converts a distance in paper coordinates to logical coordinates.
;;;
;;; (ID.DELTA-LX:CHARS LX POINT-SIZE)
;;;    Converts a logical distance to the number of fixed-width characters
;;;    of a given point size.
;;;
;;; (ID.DELTA-LY:POINTS LY)
;;; (ID.POINTS:DELTA-LY POINTS)
;;;    Converts between logical distance and points (1/72 of an inch).
;;;
;;; ID.DRAW-LINE LX1 LY1 LX2 LY2 STYLE)
;;;    Draws a line between two points.  STYLE may be one of:
;;;
;;;        NORMAL or () -- draw a thin black line.
;;;        THICK        -- draw a thick bold black line.
;;;        SHADED       -- draw a thick shaded (textured) line.
;;;        INVISIBLE    -- don't draw any line at all.
;;;
;;; (ID.DRAW-BOX LX LY WIDTH HEIGHT TEXTURE)
;;;    Draws a box whose upper left hand coordinates are LX,LY.  TEXTURE
;;;    is the texture to use to fill the inside of the box.  Currently defined
;;;      textures are (see IMPRESS-TEXTURES):
;;;
;;;        SHADED                      -- normal, even grey shading.
;;;        SHADING1                    -- same as SHADED.
;;;        SHADING2                    -- lighter shading.
;;;        SHADING3                    -- even lighter still.
;;;        SHADING4                    -- lightest.
;;;        LEFT-STRIPED-SHADING1       -- SHADING1 with left-leaning stripes.
;;;        RIGHT-STRIPED-SHADING1      -- ... right-leaning ...
;;;        LEFT-STRIPED-SHADING2       -- SHADING2 with left-leaning stripes.
;;;        RIGHT-STRIPED-SHADING2      -- ... right-leaning ...
;;;        LEFT-STRIPED-SHADING1-2     -- alternating stripes of SHADING1 and 2.
;;;          RIGHT-STRIPED-SHADING1-2    -- ...
;;;
;;; (ID.DRAW-TEXT TEXT LX LY POINT-SIZE)
;;;    Writes a text string at LX,LY (the lower left corner of the first
;;;    character).  POINT-SIZE should be the size selected by
;;;    ID.DEFINE-FONT-SIZE.
;;;
;;;=============================================================================

(defvar *id.x-min*   55.0)
(defvar *id.x-max* 2005.0)
(defvar *id.y-min*    0.0)
(defvar *id.y-max* 2625.0)
(defvar *id.paper-x-y-ratio* (// (- *id.x-max* *id.x-min*)
                                 (- *id.y-max* *id.y-min*) ) )

(defvar *id.paper:x-min&x-max&y-min&y-max*
    '( (letter 60.0 1950.0  60.0 2575.0)
       (legal  60.0 1950.0  60.0 3375.0)
       (b4     60.0 2250.0  60.0 3375.0) ) )

(defvar *id.lx-orig*  0.0)
(defvar *id.ly-orig*  0.0)
(defvar *id.lx-scale* 1.0)
(defvar *id.ly-scale* 1.0)

(defvar *id.cmasc-point-sizes* '(14 12 10 8 7) )
                        ;*** Order is significant here!
(defvar *id.cmtt-chars-per-em* 1.71)

(defvar *id.cmasc-family*    1)
(defvar *id.textures-family* 2)

(defvar *id.normal-pen-width* 1)
(defvar *id.thick-pen-width*  5)
(defvar *id.shaded-pen-width* 10)

(defvar *id.loaded-textures* () )
(declare (special *id.name:texture*) )      ;*** From IMPRE

(declare (special *im.rasters-per-inch*)

:<C.S.BULLDOG DRAWING
```

```
(defun id.start-job ( file-name paper )
    (:= *id.lx-orig*  0.0)
    (:= *id.ly-orig*  0.0)
    (:= *id.lx-scale* 1.0)
    (:= *id.ly-scale* 1.0)

    (:= *id.loaded-textures* () )

    (id.select-paper paper)

    (im.start-job file-name paper)
    (im.set-pen 1 *id.normal-pen-width*)
    () )


(defun id.select-paper ( paper-name )
    (let ( ( (name x-min x-max y-min y-max)
             (assoc paper-name *id.paper:x-min&x-max&y-min&y-max*) ) )
        (if (! name) (then
            (error (list paper-name
                         "ID.SELECT-PAPER: Invalid paper name.") ) ) )
        (:= *id.x-min* x-min)
        (:= *id.y-min* y-min)
        (:= *id.x-max* x-max)
        (:= *id.y-max* y-max)
        (:= *id.paper-x-y-ratio* (// (- *id.x-max* *id.x-min*)
                                     (- *id.y-max* *id.y-min*) ) )
        () ) )


(defun id.begin-page ()
    (im.begin-page) )

(defun id.end-page ()
    (im.end-page) )

(defun id.end-job ()
    (im.end-job) )


(defun id.define-font-size ( font-size )
    (im.create-family-table *id.cmasc-family*
                            '( (0 ,(atomconcat "cmasc" font-size) ) ) )
    (im.set-family *id.cmasc-family*)
    () )


(defun id.set-coordinates ( lx-min ly-min x-max y-max )
    (let ( (x-delta (- (flonum x-max) (flonum lx-min) ) )
           (y-delta (- (flonum y-max) (flonum ly-min) ) ) )
        (:= *id.lx-scale* (:= *id.ly-scale*
            (min (// (- *id.x-max* *id.x-min*) x-delta)
                 (// (- *id.y-max* *id.y-min*) y-delta) ) ) )

        (:= *id.lx-orig* (flonum lx-min) )
        (:= *id.ly-orig* (flonum ly-min) )
        () ) )


(defun id.set-texture ( texture-name )
    (loop (incr i from 1)
          (for (name glyph) in *id.name:texture*)
          (when (== name texture-name) )
    (do
        (if (! (member i *id.loaded-textures*) ) (then
            (im.small-glyph 0 *id.textures-family* i 32 32 0 32 32 glyph)
            (push *id.loaded-textures* i) ) )
        (im.set-texture *id.textures-family* i)
        (return () ) ) ) )


(defun id.lx:x ( lx )
    (+ (* (- (flonum lx) *id.lx-orig*)
          *id.lx-scale*)
       *id.x-min*) )

(defun id.ly:y ( ly )
    (+ (* (- (flonum ly) *id.ly-orig*)
          *id.ly-scale*)
       *id.y-min*) )


(defun id.delta-lx:delta-x ( lx )
    (* (flonum lx) *id.lx-scale*) )

(defun id.delta-ly:delta-y ( ly )
    (* (flonum ly) *id.ly-scale*) )


(defun id.delta-x:delta-lx ( x )
    (// (flonum x) *id.lx-scale*) )

(defun id.delta-y:delta-ly ( y )
    (// (flonum y) *id.ly-scale*) )



(defun id.delta-lx:chars ( lx point-size )
    (// (* (flonum lx) (* *id.lx-scale* (* 72.0 *id.cmtt-chars-per-em*) ) )
        (* (flonum *im.rasters-per-inch*) (flonum point-size) ) ) )

(defun id.delta-ly:points ( ly )
    (// (* (flonum ly) (* *id.ly-scale* 72.0) )
        (flonum *im.rasters-per-inch*) ) )

(defun id.points:delta-ly ( points )
    (// (* (flonum points) (flonum *im.rasters-per-inch*) )
        (* 72.0 *id.ly-scale*) ) )



(defun id.draw-line ( lx1 ly1 lx2 ly2 style )
    (let*( (x1   (id.lx:x lx1) )
           (y1   (id.ly:y ly1) )
           (x2   (id.lx:x lx2) )
           (y2   (id.ly:y ly2) )
           (line (id.line:clip '( (,x1 ,y1) (,x2 ,y2) ) ) ) )

        (if (&& line
                (!== 'invisible style) )
        (then
            (im.create-path line)
            (caseq style
                (thick
```

```
                (im.set-pen 1 *id.thick-pen-width*) )
            (shaded
                (im.set-pen 1 *id.shaded-pen-width*)
                (id.set-texture 'shading1) ) )
        (im.draw-path (if (== style 'shaded) 7 15) )
        (if (memq style '(shaded thick) ) (then
            (im.set-pen 1 *id.normal-pen-width*) ) ) ) )
    () ) )


(defun id.line:clip ( line )
    (&& (:= line (id.line:clip-top    line) )
        (:= line (id.line:clip-bottom line) )
        (:= line (id.line:clip-right  line) )
        (:= line (id.line:clip-left   line) ) ) )


(defun id.line:clip-top ( line )
    (let*( ( ((x1 y1) (x2 y2)) line)
           (nx1  x1)
           (ny1  y1)
           (nx2  x2)
           (ny2  y2) )
        (if (> y1 y2) (then
            (:= nx1 x2)
            (:= ny1 y2)
            (:= nx2 x1)
            (:= ny2 y1) ) )

        (? ( (>= ny1 *id.y-min*)
             line)

           ( (< ny2 *id.y-min*)
             () )

           ( (= nx1 nx2)
             '( (,nx1 ,*id.y-min*)
                (,nx1 ,ny2) ) )

           ( t
             '( ,(id.line:intersect-horizontal
                    nx1 ny1 nx2 ny2 *id.y-min*)
                (,nx2 ,ny2) ) ) ) ) )

(defun id.line:clip-bottom ( line )
    (let*( ( ((x1 y1) (x2 y2)) line)
           (nx1  x1)
           (ny1  y1)
           (nx2  x2)
           (ny2  y2) )
        (if (> y1 y2) (then
            (:= nx1 x2)
            (:= ny1 y2)
            (:= nx2 x1)
            (:= ny2 y1) ) )

        (? ( (<= ny2 *id.y-max*)
             line)

           ( (> ny1 *id.y-max*)
             () )

           ( (= nx1 nx2)
```



```
             '( (,nx1 ,ny1)
                (,nx1 ,*id.y-max*) ) )

           ( t
             '( (,nx1 ,ny1)
                ,(id.line:intersect-horizontal
                    nx1 ny1 nx2 ny2 *id.y-max*) ) ) ) ) )


(defun id.line:intersect-horizontal ( x1 y1 x2 y2 y )
    (let ( (slope  (// (- y2 y1)
                       (- x2 x1) ) ) )
        '( ,(+ x1 (// (- y y1) slope) )
           ,y) ) )


(defun id.line:clip-left ( line )
    (let*( ( ((x1 y1) (x2 y2)) line)
           (nx1  x1)
           (ny1  y1)
           (nx2  x2)
           (ny2  y2) )
        (if (> x1 x2) (then
            (:= nx1 x2)
            (:= ny1 y2)
            (:= nx2 x1)
            (:= ny2 y1) ) )

        (? ( (>= nx1 *id.x-min*)
             line)

           ( (< nx2 *id.x-min*)
             () )

           ( (= ny1 ny2)
             '( (,*id.x-min* ,ny1)
                (,nx2        ,ny1) ) )

           ( t
             '( ,(id.line:intersect-vertical
                    nx1 ny1 nx2 ny2 *id.x-min*)
                (,nx2 ,ny2) ) ) ) ) )


(defun id.line:clip-right ( line )
    (let*( ( ((x1 y1) (x2 y2)) line)
           (nx1  x1)
           (ny1  y1)
           (nx2  x2)
           (ny2  y2) )
        (if (> x1 x2) (then
            (:= nx1 x2)
            (:= ny1 y2)
            (:= nx2 x1)
            (:= ny2 y1) ) )

        (? ( (<= nx2 *id.x-max*)
             line)

           ( (> nx1 *id.x-max*)
             () )
```

```
        ( (= ny1 ny2)
          '( (,nx1        ,ny1)
             (,*id.x-max* ,ny1) ) )

        ( t
          '( (,nx1 ,ny1)
             ,(id.line:intersect-vertical
                 nx1 ny1 nx2 ny2 *id.x-max*) ) ) ) ) )


(defun id.line:intersect-vertical ( x1 y1 x2 y2 x )
    (let ( (slope  (// (- y2 y1)
                       (- x2 x1) ) ) )
        '( ,x
           ,(+ y1 (* slope (- x x1) ) ) ) ) ) )



(defun id.draw-box ( lx ly width height texture )
    (let*( (x1  (id.lx:x lx) )
           (x2  (+ (id.lx:x lx) (id.delta-lx:delta-x width) ) )
           (y1  (id.ly:y ly) )
           (y2  (+ (id.ly:y ly) (id.delta-ly:delta-y height) ) )
           (box (id.box:clip '(,x1 ,y1 ,x2 ,y2) ) ) )

        (if box (then
            (let ( ( (x1 y1 x2 y2) box) )
                (im.create-path '( (,x1 ,y1)
                                   (,x1 ,y2)
                                   (,x2 ,y2)
                                   (,x2 ,y1)
                                   (,x1 ,y1) ) )
                (im.draw-path 15)
                (if texture (then
                    (id.set-texture texture)
                    (im.fill-path 7) ) ) ) ) )
        () ) )


(defun id.box:clip ( box )
    (&& (:= box (id.box:clip-top    box) )
        (:= box (id.box:clip-bottom box) )
        (:= box (id.box:clip-left   box) )
        (:= box (id.box:clip-right  box) ) ) )


(defun id.box:clip-top ( box )
    (let ( ( (x1 y1 x2 y2) box) )

        (? ( (>= y1 *id.y-min*)
             box)
           ( (< y2 *id.y-min*)
             () )
           ( t
             '(,x1 ,*id.y-min* ,x2 ,y2) ) ) ) )

(defun id.box:clip-bottom ( box )
    (let ( ( (x1 y1 x2 y2) box) )

        (? ( (<= y2 *id.y-max*)
             box)
           ( (> y1 *id.y-max*)
```



```
             () )
           ( t
             '(,x1 ,y1 ,x2 ,*id.y-max*) ) ) ) )

(defun id.box:clip-left ( box )
    (let ( ( (x1 y1 x2 y2) box) )

        (? ( (>= x1 *id.x-min*)
             box)
           ( (< x2 *id.x-min*)
             () )
           ( t
             '(,*id.x-min* ,y1 ,x2 ,y2) ) ) ) )

(defun id.box:clip-right ( box )
    (let ( ( (x1 y1 x2 y2) box) )

        (? ( (<= x2 *id.x-max*)
             box)
           ( (> x1 *id.x-max*)
             () )
           ( t
             '(,x1 ,y1 ,*id.x-max* ,y2) ) ) ) )



(defun id.draw-text ( text lx ly point-size )
    (let ( (blx (id.lx:x lx) )
           (bly (id.ly:y ly) ) )
    (loop (for char in (if (consp text) text (aexplodec text) ) )
          (initial dx (// (* (flonum point-size)
                             (flonum *im.rasters-per-inch*) )
                          (* *id.cmtt-chars-per-em* 72.0) )
                   dy (// (* (flonum point-size)
                             (flonum *im.rasters-per-inch*) )
                          72.0)
                   try (- bly dy)
                   trx      0.0
                   set-x-y? () )
    (next trx (+ blx dx) )
    (do
        (if (id.box:clip '(,blx ,try ,trx ,bly) ) (then
            (if (! set-x-y?) (then
                (im.set-xpos blx)
                (im.set-ypos bly)
                (:= set-x-y? t) ) )
            (if (!= #/  char) (then
                (im.write-8 char) )
            (else
                (im.set-xpos-relative dx) ) ) ) ) )
    (next blx trx) ) ) )
```

```
(eval-when (compile)
    (build '(drawing:impress) ) )

(declare (special *id.shading1*) )
(:= *id.shading1* '#.(im.texture:read)
@...@...@...@...@...@...@...@...
................................
..@...@...@...@...@...@...@...@.
................................
@...@...@...@...@...@...@...@...
................................
..@...@...@...@...@...@...@...@.
................................
@...@...@...@...@...@...@...@...
................................
..@...@...@...@...@...@...@...@.
................................
@...@...@...@...@...@...@...@...
................................
..@...@...@...@...@...@...@...@.
................................
@...@...@...@...@...@...@...@...
................................
..@...@...@...@...@...@...@...@.
................................
@...@...@...@...@...@...@...@...
................................
..@...@...@...@...@...@...@...@.
................................
@...@...@...@...@...@...@...@...
................................
..@...@...@...@...@...@...@...@.
................................
)


(declare (special *id.shading2*) )
(:= *id.shading2* '#.(im.texture:read)
@.......@.......@.......@.......
................................
....@.......@.......@.......@...
................................
@.......@.......@.......@.......
................................
....@.......@.......@.......@...
................................
@.......@.......@.......@.......
................................
....@.......@.......@.......@...
................................
@.......@.......@.......@.......
................................
....@.......@.......@.......@...
................................
@.......@.......@.......@.......
................................
....@.......@.......@.......@...
................................
@.......@.......@.......@.......
................................
....@.......@.......@.......@...
................................
@.......@.......@.......@.......
```

1

```
................................
....@.......@.......@.......@...
................................
@.......@.......@.......@.......
................................
....@.......@.......@.......@...
................................
@.......@.......@.......@.......
................................
....@.......@.......@.......@...
................................
@.......@.......@.......@.......
)


(declare (special *id.shading3*) )
(:= *id.shading3* '#.(im.texture:read)
@.......@.......@.......@.......
................................
................................
................................
....@.......@.......@.......@...
................................
................................
................................
@.......@.......@.......@.......
................................
................................
................................
....@.......@.......@.......@...
................................
................................
................................
@.......@.......@.......@.......
................................
................................
................................
....@.......@.......@.......@...
................................
................................
................................
@.......@.......@.......@.......
................................
................................
................................
....@.......@.......@.......@...
................................
................................
................................
)


(declare (special *id.shading4*) )
(:= *id.shading4* '#.(im.texture:read)
@...............@...............
................................
................................
................................
........@...............@.......
................................
................................
................................
@...............@...............
```

2

PS:<C.S.BULLDOG.DRAWING>IMPRESS-TEXTURES.LSP.7

```
.............................
.............................
.............................
........@.............@.......
.............................
.............................
.............................
@.............@.............
.............................
.............................
.............................
.............................
........@.............@.......
.............................
.............................
.............................
@.............@.............
.............................
.............................
.............................
........@.............@.......
.............................
.............................
.............................
)


(declare (special *id.left-stripes*) )
(:= *id.left-stripes* '#.(im.texture:read)
@@@@@@@@@@@@@@@@.............
.@@@@@@@@@@@@@@@@............
..@@@@@@@@@@@@@@@@...........
...@@@@@@@@@@@@@@@@..........
....@@@@@@@@@@@@@@@@.........
.....@@@@@@@@@@@@@@@@........
......@@@@@@@@@@@@@@@@.......
.......@@@@@@@@@@@@@@@@......
........@@@@@@@@@@@@@@@@.....
.........@@@@@@@@@@@@@@@@....
..........@@@@@@@@@@@@@@@@...
...........@@@@@@@@@@@@@@@@..
............@@@@@@@@@@@@@@@@.
.............@@@@@@@@@@@@@@@@
@.............@@@@@@@@@@@@@@@
@@.............@@@@@@@@@@@@@@
@@@.............@@@@@@@@@@@@@
@@@@.............@@@@@@@@@@@@
@@@@@.............@@@@@@@@@@@
@@@@@@.............@@@@@@@@@@
@@@@@@@.............@@@@@@@@@
@@@@@@@@.............@@@@@@@@
@@@@@@@@@.............@@@@@@@
@@@@@@@@@@.............@@@@@@
@@@@@@@@@@@.............@@@@@
@@@@@@@@@@@@.............@@@@
@@@@@@@@@@@@@.............@@@
@@@@@@@@@@@@@@.............@@
@@@@@@@@@@@@@@@.............@
)
```

3

```
(declare (special *id.right-stripes*) )
(:= *id.right-stripes* '#.(im.texture:read)
@@@@@@@@@@@@@@@@.............
@@@@@@@@@@@@@@@@.............@
@@@@@@@@@@@@@@@@............@@
@@@@@@@@@@@@@@@@...........@@@
@@@@@@@@@@@@@@@@..........@@@@
@@@@@@@@@@@@@@@@.........@@@@@
@@@@@@@@@@@@@@@@........@@@@@@
@@@@@@@@@@@@@@@@.......@@@@@@@
@@@@@@@@@@@@@@@@......@@@@@@@@
@@@@@@@@@@@@@@@@.....@@@@@@@@@
@@@@@@@@@@@@@@@@....@@@@@@@@@@
@@@@@@@@@@@@@@@@...@@@@@@@@@@@
@@@@@@@@@@@@@@@@..@@@@@@@@@@@@
@@@@@@@@@@@@@@@@.@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@.
@@@@@@@@@@@@@@@@@@@@@@@@@@@@..
@@@@@@@@@@@@@@@@@@@@@@@@@@@...
@@@@@@@@@@@@@@@@@@@@@@@@@@....
@@@@@@@@@@@@@@@@@@@@@@@@@.....
@@@@@@@@@@@@@@@@@@@@@@@@......
@@@@@@@@@@@@@@@@@@@@@@@.......
@@@@@@@@@@@@@@@@@@@@@@........
@@@@@@@@@@@@@@@@@@@@@.........
@@@@@@@@@@@@@@@@@@@@..........
@@@@@@@@@@@@@@@@@@@...........
@@@@@@@@@@@@@@@@@@............
@@@@@@@@@@@@@@@@@.............
)


(declare (special *id.left-striped-shading1*) )
(:= *id.left-striped-shading1*
    (im.glyph:logand *id.left-stripes*
                     *id.shading1*) )

(declare (special *id.right-striped-shading1*) )
(:= *id.right-striped-shading1*
    (im.glyph:logand *id.right-stripes*
                     *id.shading1*) )

(declare (special *id.left-striped-shading2*) )
(:= *id.left-striped-shading2*
    (im.glyph:logand *id.left-stripes*
                     *id.shading2*) )

(declare (special *id.right-striped-shading2*) )
(:= *id.right-striped-shading2*
    (im.glyph:logand *id.right-stripes*
                     *id.shading2*) )

; (:= *id.left-striped-shading1-2*
;     (im.glyph:logior
;         (im.glyph:logand *id.left-stripes*
;                          *id.shading1*)
;         (im.glyph:logand (im.glyph:lognot *id.left-stripes*)
```

4

```
                        *id.shading2*) ) )
;
; The above produces little glitches which I've cleaned up by hand:
;
(declare (special *id.left-striped-shading1-2*) )
(:= *id.left-striped-shading1-2* '#.(im.texture:read)
@...@...@...@...@.......@.......
................................
..@...@...@...@.....@.......@...
................................
@...@...@...@...@.......@.......
................................
......@...@...@...@.........@...
................................
@.......@...@...@...@...@.......
................................
....@.....@...@...@...@.....@...
................................
@.......@...@...@...@...@.......
................................
....@.........@...@...@...@.....
................................
@.......@.......@...@...@...@...
................................
....@.......@.....@...@...@...@.
................................
@.......@.......@...@...@...@...
................................
..@.........@.........@...@...@.
................................
@...@...@.......@.......@...@...
................................
..@...@.....@.......@.....@...@.
................................
@...@...@.......@.......@...@...
................................
..@...@...@.........@.........@.
................................
)


; (:= *id.right-striped-shading1-2*
;     (im.glyph:logior
;         (im.glyph:logand *id.right-stripes*
;                          *id.shading1*)
;         (im.glyph:logand (im.glyph:lognot *id.right-stripes*)
;                          *id.shading2*) ) )
;
; The above produces little glitches which I've cleaned up by hand:
;
(declare (special *id.right-striped-shading1-2*) )
(:= *id.right-striped-shading1-2* '#.(im.texture:read)
@...@...@...@...@.......@.......
................................
..@...@...@.........@.........@.
................................
@...@...@.......@.......@...@...
................................
..@...@.....@.......@.....@...@.
................................
@...@...@.......@.......@...@...
................................
..@.........@.........@.........@.
................................
```

5

```
................................
@.......@.......@...@...@...@...
................................
....@.......@.....@...@...@...@.
................................
@.......@.......@...@...@...@...
................................
....@.....@...@...@...@.....@...
................................
@.......@...@...@...@...@.......
................................
......@...@...@...@.........@...
................................
@...@...@...@...@.......@.......
................................
..@...@...@...@.....@.......@...
................................
)


(declare (special *id.name:texture*) )
(:= *id.name:texture*
    '( (shaded                      ,*id.shading1*)
       (shading1                    ,*id.shading1*)
       (shading2                    ,*id.shading2*)
       (shading3                    ,*id.shading3*)
       (shading4                    ,*id.shading4*)
       (left-striped-shading1       ,*id.left-striped-shading1*)
       (right-striped-shading1      ,*id.right-striped-shading1*)
       (left-striped-shading2       ,*id.left-striped-shading2*)
       (right-striped-shading2      ,*id.right-striped-shading2*)
       (left-striped-shading1-2     ,*id.left-striped-shading1-2*)
       (right-striped-shading1-2 ,*id.right-striped-shading1-2*) ) )

(defun it.dummy ( x ) x)    ;*** Force this file to compile.
```

6

```
;===============================================================================
;
; WRITE-MIS
;
; This module implements the DAG-drawing interface to the compiler that
; draws out the MI-flow-graph.
;
; To draw a DAG of an MI, load this module BEFORE running SKEX.  Trace
; scheduling will stop with a breakpoint twice for each trace:  after
; the trace has been picked but before it has been scheduled, and after
; the bookkeeping is done.  To print the flow graph at any breakpoint,
; do:
;
;     (write-mis)
;
; This writes out a description of the flow graph as a DAG to the file
; IN.DAG.  Only the current trace MIs and and their immediate neighbors
; will be written; to write out the whole flow-graph:
;
;     (write-mis () )
;
; Now switch to another Lisp process that has DRAWING:BUILD.LSP loaded
; into it (you need another Lisp because the dag-drawing algorithms use
; piggy datastructures).  In that Lisp, do:
;
;     (dag:impress)
;
; This reads in the DAG stored in IN.DAG and writes out the file DAG.IMPA
; which can then be printed on the Imagen.  See WRITE-TRACE.LSP for a
; fuller discussion of DAG:IMPRESS.
;
; (WRITE-MIS &OPTIONAL (TRACE-ONLY? T) (FILENAME "IN.DAG") DAG-COMMANDS)
;
;     This writes out a snapshot of the current MI-flow-graph in a form
;     suitable for the dag-drawing functions.  TRACE-ONLY?, which defaults
;     to T, says whether to print only the current trace and immediate
;     neighbors or the whole MI-flow-graph.  FILENAME is the optional
;     name of the output file to write the DAG to.  DAG-COMMANDS is an
;     optional list of assignment statements that will be recorded with
;     the DAG and EVALed by the DAG-drawing software; these assignments
;     control the formatting of the DAG.  See the variable
;     *WM.DEFAULT-DAG-COMMANDS* below; DAG-DECLARATIONS.LSP contains
;     description of user-settable formatting variables that can be
;     changed.
;
;===============================================================================


(eval-when (compile load)
    (include trace:declarations) )

(eval-when (compile)
    (build '(
        utilities:bit-set
        utilities:sharp-sharp) ) )
(eval-when (compile eval)
    (build '(drawing:dag-input-node) ) )


(declare (special
    *wm.trace-only?*
    *wm.back-edges*
    *wm.mi:descendants*
```



```
    *wm.default-dag-commands*

    *hash-table.not-found*
    *bit-set.empty-set*
    ) )


(:= *tr.break-before-scheduling* t)
(:= *tr.break-after-bookkeeping* t)


(:= *wm.default-dag-commands*
    '( (:= *dag:box-height*                   46)
       (:= *dag:box-width*                   180)
       (:= *dag:node-width*                  200)
       (:= *dag:line-width*                   25)
       (:= *dag:minimum-level-separation*     20)
       (:= *dag:ideal-text-lines-per-node* 1000)
       (:= *dag:critical-threshold*           0)
       (:= *dag:remove-non-critical-nodes&edges?* () )
       (:= *dag:shade-critical-nodes?*        () )
       (:= *dag:text-x-margin*                8)
       (:= *dag:make-slides*?                 () ) ) )


(defun write-mis ( &optional (*wm.trace-only?* t)
                             (file-name "in.dag")
                             dag-commands )

    (iota ( (file file-name '(out newversion) ) ) (without file
        (let ( (*wm.back-edges* () )
               (*wm.mi:descendants* (hash-table:create) ) )

            (if (! (boundp '*tr.trace-number*) ) (then
                (:= *tr.trace-number* 0) ) )
            (wm.set-descendants *tr.s*)
            (wm.set-back-edges  *tr.s*)

            (msg (h '(,,*wm.default-dag-commands* ,,dag-commands)
                    100000 100000) t)
            (msg (h (wm.mis:graph *tr.s*) 100000 100000) t)

            (filename file) ) ) ) )


(defun wm.set-descendants ( mis )
    (loop (for mi in mis)
          (when (! (mi:preds mi) ) )
    (do
        (wm.mi:set-descendants-visit mi *bit-set.empty-set*) ) )
    () )


(defun wm.mi:set-descendants-visit ( mi ancestors )
    (let ( (current-descendants (wm.mi:descendants mi) ) )

        (if (! (bit-set:= current-descendants *bit-set.empty-set*) ) (then
            current-descendants)

        (else
            (:= ancestors (bit-set:unioni ancestors (mi:number mi) ) )
            (loop (for succ-mi in (mi:succs mi) )
                  (initial descendants (bit-set:singleton (mi:number mi) ) )
```

```
            (do
                (if (! (bit-set:member? ancestors (mi:number succ-mi) ) )
                (then
                    (:= descendants
                        (bit-set:union descendants
                            (wm.mi:set-descendants-visit succ-mi
                                                         ancestors))))))
            (result (wm.mi:set-descendants mi descendants) ) ) ) ) ) )

(defun wm.mi:descendants ( mi )
    (let ( (result (hash-table:get *wm.mi:descendants* mi) ) )
        (if (== *hash-table.not-found* result) (then
            *bit-set.empty-set*)
        (else
            result) ) ) )

(defun wm.mi:set-descendants ( mi descendants )
    (hash-table:put *wm.mi:descendants* mi descendants) )


(defun wm.mi:ancestor? ( mi1 mi2 )
    (bit-set:member? (wm.mi:descendants mi1)
                     (mi:number mi2) ) )



(defun wm.set-back-edges ( mis )
    (loop (for mi in mis) (do
        (loop (for succ-mi in (mi:succs mi) ) (do
            (if (wm.mi:ancestor? succ-mi mi) (then
                (push *wm.back-edges* '(,mi ,succ-mi) ) ) ) ) ) ) )
    () )


(defun wm.mi:back-edge? ( tail-mi head-mi )
    (loop (for (t-mi h-mi) in *wm.back-edges*) (do
        (if (&& (== t-mi tail-mi)
                (== h-mi head-mi) )
        (then
            (return t) ) ) )
    (result () ) ) )


(defun wm.mis:graph ( mis )
    (loop (for mi in mis)
          (when (wm.mi:interesting? mi) )
          (initial graph () )
    (do
        (push graph
              (dag:input-node:new
                name              (mi:number mi)
                child-edge-styles (wm.mi:graph-child-edge-styles mi)
                label             (wm.mi:label mi)
                style             (wm.mi:style mi) ) ) )
    (result graph) ) )


(defun wm.mi:interesting? ( mi )
    (|| (! *wm.trace-only?*)
        (wm.mi:on-trace? mi)

        (loop (for neighbor-mi in (append (mi:succs mi)
                                          (mi:preds mi) ) )
```



```
        (do
            (if (wm.mi:on-trace? neighbor-mi) (then
                (return t) ) ) )
        (result () ) ) ) )


(defun wm.mi:on-trace? ( mi )
    (|| (= *tr.trace-number* (mi:trace mi) )
        (&& (! (mi:compacted? mi) )
            (mi:trace-pos mi) ) ) )



(defun wm.mi:graph-child-edge-styles ( mi )
    (let ( (child-edge-styles () ) )

        (loop (for succ-mi in (mi:succs mi) )
              (incr i from 1)
              (when (wm.mi:interesting? succ-mi) )
              (when (! (wm.mi:back-edge? mi succ-mi) ) )
        (do
           (push child-edge-styles
                 '(,(mi:number succ-mi)
                   ,(if (= i 1) 'normal 'thick) ) ) ) )

        (loop (for pred-mi in (mi:preds mi) )
              (initial tail-mi       ()
                       head-mi       ()
                       head-succ-mis () )
              (when (wm.mi:interesting? pred-mi) )
              (when (wm.mi:back-edge? pred-mi mi) )
        (do
            (push child-edge-styles
                  '(,(mi:number pred-mi)
                    shaded) ) ) )

        child-edge-styles) )



(defun wm.mi:label ( mi )
    '( (,(mi:number mi) ,,(&& (mi:trace mi)
                              '(trace ,(mi:trace mi) ) ) )
       ,(mi:source mi) ) )


(defun wm.mi:style ( mi )
    (if (! (mi:compacted? mi) ) (then
        (? ( (mi:trace-pos mi)
             'shading1)
           ( (&& (= *tr.trace-number* (mi:trace mi) )
                 (== 'join (mi:copy-type mi) ) )
             'right-striped-shading2)
           ( (&& (= *tr.trace-number* (mi:trace mi) )
                 (== 'split (mi:copy-type mi) ) )
             'left-striped-shading2)
           ( t
             () ) ) )
    (else
        (if (= *tr.trace-number* (mi:trace mi) )
            'shading1
            () ) ) ) )
```

```
;===============================================================================
;
; WRITE-TRACE
;
; This module provides the interface functions for printing out the DAGs
; representing the dataprecedence graph of individual traces.  To print
; out the DAG for a trace, load this module BEFORE running SKEX.  During
; trace-scheduling, there will be a breakpoint after each trace is
; scheduled.  If you wish to print the corresponding DAG, do:
;
;     (write-trace)
;
; This writes a file IN.DAG.  Now switch to another Lisp with
; DRAWING:BUILD.LSP loaded in it.  Do:
;
;     (dag:impress)
;
; This reads in IN.DAG and writes a file DAG.IMPA, which can then be printed
; on the Imagen.
;
; (DAG:IMPRESS &OPTIONAL (INFILE "IN.DAG") (OUTFILE "DAG.IMPA") )
;
;     Reads in a DAG description from INFILE and pretty prints it to the
;     ImPress file OUTFILE.  There are a number of parameters that control
;     the formatting of the DAG; see DRAWING:DAG-DECLARATIONS.LSP.  Those
;     paramaters can be changed by passing them in with the function
;     WRITE-TRACE below.
;
; (WRITE-TRACE &OPTIONAL (FILENAME "IN.DAG") DAG-COMMANDS)
;
;     Writes a description of the current trace to FILENAME.  DAG-COMMANDS
;     is an optional list of assignment statements that will be recorded
;     with the DAG and EVALed by the DAG-drawing software; these
;     assignments control the formatting of the DAG.  See the variable
;     *WT.DEFAULT-DAG-COMMANDS* below.
;
;===============================================================================


(eval-when (compile load)
    (include trace:declarations) )
(eval-when (compile)
    (build '(
        ideal-code-generator:schedule) ) )
(eval-when (compile eval)
    (build '(drawing:dag-input-node) ) )

(declare (special
    *wt.current-trace*

    *tr.dag-hook*
    ) )

(defvar *wt.default-dag-commands*
    '( (:= *dag:box-height*                46)
       (:= *dag:node-width*                 100)
       (:= *dag:line-width*                 25)
       (:= *dag:box-width*                  80)
       (:= *dag:minimum-level-separation*       30)
       (:= *dag:ideal-text-lines-per-node* 4)
       (:= *dag:critical-threshold*         0)
       (:= *dag:remove-non-critical-nodes&edges?* () )
       (:= *dag:shade-critical-nodes?*      t)
```

1

```
       (:= *dag:text-x-margin*              3)
       (:= *dag:make-slides*?               () ) ) )


(defun wt.dag-hook ( trace )
    (if (! trace) (then
        (:= *wt.current-trace* () ) )
    (else
        (:= *wt.current-trace* trace)
        (break-point after-trace-picked) ) ) )


(:= *tr.dag-hook* 'wt.dag-hook)


(defun write-trace ( &optional (file-name "in.dag") dag-commands)
    (iota ( (file file-name '(out newversion) ) ) (without file
        (msg (h `(,,*wt.default-dag-commands* ,,dag-commands)
                100000 100000) t)
        (msg (h (wt.trace:graph *wt.current-trace*) 100000 100000) t)
        (filename file) ) ) )


(defun wt.trace:graph ( trace )
    (for (elem in trace) (save
        (dag:input-node:new
            name
                (trace-element:name elem)
            child-edge-styles
                (for (child-elem in (trace-element:successors elem) )
                     (reason     in (trace-element:reasons    elem) )
                (save
                    `(,(trace-element:name   child-elem)
                      ,(wt.reason:edge-style reason) ) ) )
            label
                `(,(trace-element:name   elem)
                  ,(trace-element:source elem) )
            style
                () ) ) ) )


(defun trace-element:name ( elem )
    (mi:number (trace-element:bookkeeper-record elem) ) )

(defun wt.reason:edge-style ( reason )
    (caseq reason
        (possible-operand-conflict 'shaded)
        (conditional-conflict      'thick)
        (t                         'normal) ) )
```

2

PS:<C.S.BULLDOG.DRAWING>WRITE-TRACE.LSP.7