The MLgraph Primer

Emmanuel CHAILLOUX[*,**] Guy COUSINEAU[*]

[*]LIENS, URA 1327 du CNRS
École Normale Supérieure
[**]LITP, URA 248 du CNRS
Université Pierre et Marie Curie - Paris 6

LIENS - 92 - 15

September 1992

The MLgraph Primer

Emmanuel CHAILLOUX[1,2] – Guy COUSINEAU[1]

About MLgraph

This documents describes the MLgraph system, a library for producing images in ML. This library is currently available via ftp anonymous as an extension to Caml Light [9].

The graphical model which is used in MLgraph is basically that of PostScript [2],[1]. Various objects can be defined on the infinite cartesian plane and arbitrarily scaled, translated and rotated by the application of linear transformations. Each category of graphical objects corresponds to an ML type. A type `picture` is used to represent all printable objects. Pictures have a "frame" and possibly a set of named "handles" that are used for combination operations. Pictures are defined from more basic objects such as geometric elements (lines, arcs and curves), texts and bitmaps. All operations defined on these types are purely functional except for pixel editing in bitmaps.

Printing is obtained via a translation to PostScript. The philosophy of this translation has been to delegate as much work as possible to PostScript. In particular, all applications of linear transformations to pictures are delegated to the PostScript interpretor. This has two advantages: the efficiency of PostScript interpretors is fully used and the sharing involved in the ML representation of pictures is preserved as much as possible.

About this Manual

We have tried to keep this manual small and therefore, it is not strictly self-contained. In particular some PostScript notions are used but not explained. The reader which is not familiar with PostScript might have, at some points, to refer to the PostScript Reference manual [2].

This manual is divided into three sections. The first section is a general presentation of MLgraph. The second section details two extended examples. The third section explains how to install the system and put it to real use. This manual also contains a glossary of MLgraph functions with their types and an index of MLgraph notions .

[1] URA 1327 - Laboratoire d'Informatique de l' École Normale Supérieure - 45 rue d'Ulm, 75230 Paris Cédex 05, France. Electronic mail: Emmanuel.Chailloux@ens.fr, Guy.Cousineau@ens.fr

[2] URA 248 - Laboratoire d'Informatique Théorique et Programmation - Institut Blaise Pascal - 4, place Jussieu - UPMC - 75252 Paris Cédex 05, France. Electronic mail : ec@litp.ibp.fr

Acknowledgments

We are grateful to Yves Lafont and Pierre Crégut who have been the first MLgraph users and made many useful suggestions. We also thank Michel Mauny for his comments on a first version of this text.

Chapter 1

Description of the System

1.1 Basic Geometric Notions

Graphic objects are arbitrarily located in the cartesian plane a small part of which is shown in figure 1.1. A point in this plane is represented by a Caml Light object of type `point` defined by:

```
type point = {xc:float;yc:float}
```

`origin:point` is the origin of the coordinate system. Points coordinates are expressed in typographic points (1/72 inch).

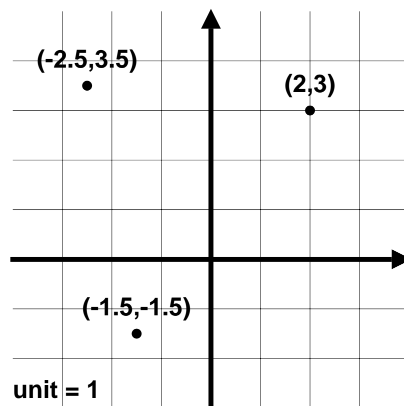


Figure 1.1: Points in the cartesian plane

1.1.1 Geometric elements

A geometric element is either a polygonal line represented by a sequence of points or a circle arc represented by a center, a radius and two angles or a Beziers curve represented by a start point, two control points, and an end point. The corresponding type is the following:

```
type point = {xc:float;yc:float};;
type geom_element =
  Seg of point list
| Arc of point * float * float * float
| Curve of point * point * point * point;;
```

Given the points A=(-3.,-3.), B=(-3.,-1.), C=(-1.,-1.), D=(-1.,-3.), E=(-3.,-4.), F=(-2.,0.), G=(0.,5.), H=(1.,4.) and I=(3.,0.), the expressions Seg [A; B; C; D; A], Curve(E, F, G, H) and Arc(I, 2., 30., 290.) correspond to the the three elements that are drawn with thick lines in figure 1.2.

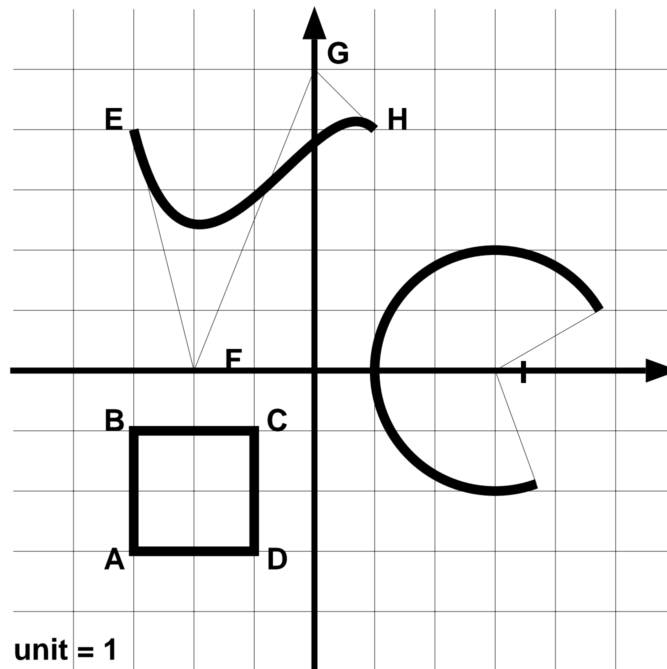


Figure 1.2: Geometric elements

1.1.2 Transformations

A transformation is represented by a 3×3 matrix which has shape:

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix}$$

It operates on vectors

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

that represent points with coordinates (x,y).

The library offers various functions to build transformations:

```
id_trans : transformation
translation : float * float -> transformation
rotation : point -> float -> transformation
scaling : float * float -> transformation
line_symmetry : point * point -> transformation
point_symmetry : point -> transformation
handle_transform : point * point -> point * point -> transformation
make_transformation : float * float * float * float * float * float ->
transformation
```

- `id_trans` is the identity transformation.
- `translation(a,b)` builds a transformation which performs displacement `a` on the x-coordinate and displacement `b` on the y-coordinate.
- `rotation pt alpha` builds a rotation with center `pt` and angle `alpha`. Angles are expressed in degrees.
- `scaling (a,b)` builds a scaling transformation which uses scale factor `a` on the x-coordinate and scale factor `b` on y-coordinate.
- `line_symmetry (p1,p2)` builds a symmetry according to the line defined by the two points `p1` and `p2`. It fails if the two points are equal.
- `point_symmetry p` builds a symmetry according to point `p`.
- `handle_transform (pt1,pt2) (pt3,pt4)` builds a linear transformation that maps `pt1` to `pt3` and `pt2` to `pt4`. It fails if `pt1`=`pt2` or `pt3`=`pt4`. Normally, the user will use this function only implicitly when performing higher level picture composition. (cf. section 1.5.3).
- `make_transformation(a,b,c,d,e,f)` creates a transformation with matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix}$$

The available operations on transformations are the followings:

```
compose_transformations : transformation list -> transformation
prefix ctrans : transformation -> transformation -> transformation
inverse_transformation : transformation -> transformation
```

- `compose_transformations tl` produces the composition of transformation list `tl`, computed from right to left.
- `ctrans` is an infix binary composition.
- `inverse_transformation T` produces the inverse of transformation T provided that it is invertible.

1.2 Sketches and Pictures

1.2.1 Sketches

A sketch is basically a sequence of geometric elements. It corresponds roughly to what is called a "path" in the PostScript terminology.

Sketches are represented by a Caml Light type `sketch` which is used as an abstract type. The representation of type `sketch` involves lists of geometric elements together with additional information such as frame (cf. 1.2.5) and interface information (cf. 1.5.3).

The basic building function for sketches is:

```
make_sketch : geom_element list -> sketch
```

The sketches built by function `make_sketch` are connected sets of geometric elements. When two consecutive geometric elements in the list are disconnected (i.e. the end point of the first one does not match the start point of the second one), then a line is added to connect them. Therefore, given four points A, B ,C, D, the two expressions `make_sketch [Seg[A;B;C;D]]` and `make_sketch [Seg[A;B];Seg[C;D]]` define equivalent sketches.

It is however possible to make sketches which are formed of several disconnected parts by using function

```
group_sketches : sketch list -> sketch
```

The two expressions `make_sketch [Seg[A;B];Seg[C;D]]` and `group_sketches [make_sketch [Seg[A;B]]; make_sketch [Seg[C;D]]]` correspond to non equivalent sketches since line BC exists in the first but not in the second.

Figure 1.3 shows a text defining a sketch together with its graphical representation.

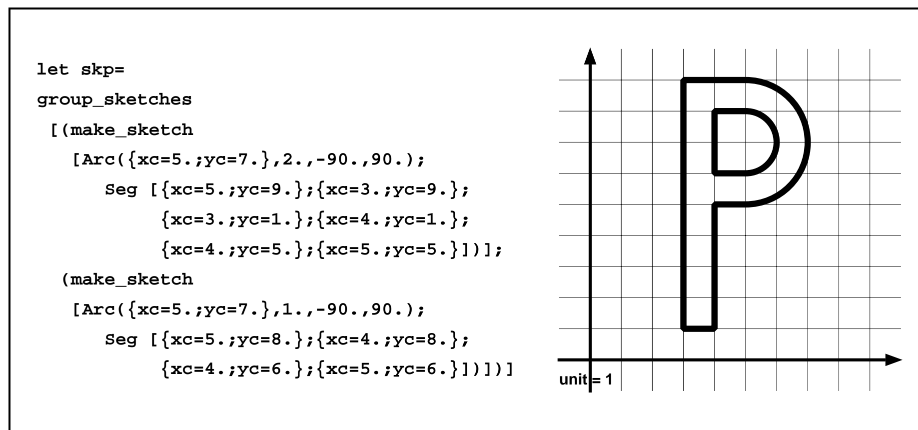


Figure 1.3: A sketch representing letter P

1.2.2 Painting Information

Sketches are in fact made of pure lines. To be visualized as images, they must be transformed into pictures using painting information. This information is represented by values of the following types.

```
type linecap = Buttcap | Squarecap | Roundcap;;
type linejoin = Beveljoin | Roundjoin | Miterjoin;;
type linestyle = {linewidth:float;
                  linecap:linecap;
                  linejoin:linejoin;
                  dashpattern:int list};;
type fillstyle = Nzfill | Eofill;;
type clipstyle  = Nzclip | Eoclip;;
type color = Rgb of float * float * float
           | Hsb of float * float * float
           | Gra of float;;
```

We do not detail here the meaning of these types which refer exactly to PostScript notions.

1.2.3 Pictures

The type `picture` is used for all visual objects in the system. Pictures can be built from sketches using painting information. As will be shown later, they can also be built from bitmaps and texts.

Here are the three main functions that make pictures from sketches:

```
make_draw_picture : linestyle * color -> sketch -> picture
make_closed_draw_picture : linestyle * color -> sketch -> picture
make_fill_picture : fillstyle * color -> sketch -> picture
```

- `make_draw_picture` produces a picture from a sketch by giving uniformly a linestyle and a color to its elements.
- `make_closed_draw_picture` is similar to `make_draw_picture` but it closes each of its connected parts as does `make_fill_picture`. Closing the connected parts of a sketch means making sure that each connected part ends at its starting point. It also makes line joins correct at this point.
- `make_fill_picture` produces a picture from a sketch by first closing each of its connected parts (those that were obtained by function `make_sketch`). This is done by adding a line from the start point to the end point. Then the "interior" of the sketch is filled according to the given fillstyle with the paint of the given color.

Given the sketch `skp` described in figure 1.3, it is possible to define how to draw it or to fill it as shown in figure 1.4.
It is also possible to use a sketch to "clip" part of a picture.

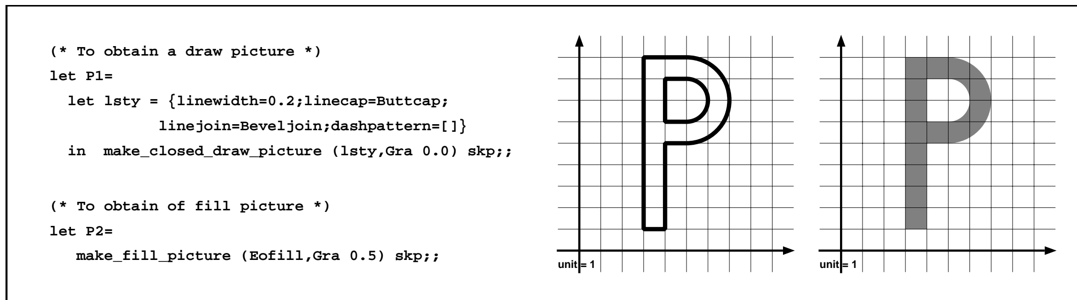


Figure 1.4: Two pictures obtained from the same sketch

```
clip_picture : clipstyle -> sketch -> picture -> picture
```

- `clip_picture clipsty sk pict` builds a picture containing the part of the picture `pict` which is in the "interior" of sketch `sk` according to clipping rule `clipsty`. As for function `make_fill_picture`, the interior of sketch `sk` is computed after closing its connected parts.

Pictures can be grouped together using the function :

```
group_pictures : picture list -> picture
```

The grouping is performed from left to right. Each new picture can cover previous ones partially or totally. Figure 1.5 shows what is obtained by grouping the two pictures of figure 1.4.

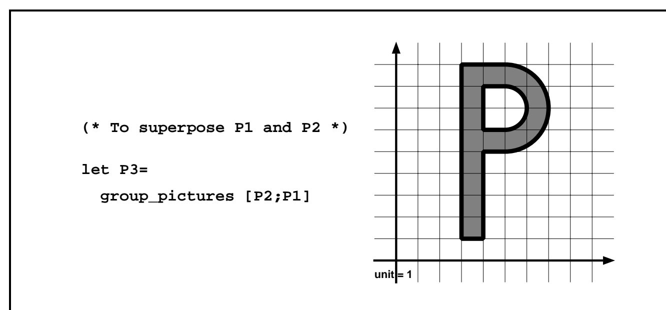


Figure 1.5: A superposition of pictures P1 and P2

1.2.4 Applying transformations to sketches and pictures

Transformations are applied to sketches and pictures using functions

```
transform_sketch : transformation -> sketch -> sketch
transform_picture : transformation -> picture -> picture
```

Figure 1.6 describes the effect of applying successively transformations `T1= translation (2.,-9.)`, `T2= scaling (0.5,0.5)` and `T3= rotation {xc=-2.;yc=-2.} 60.0` to our basic picture.

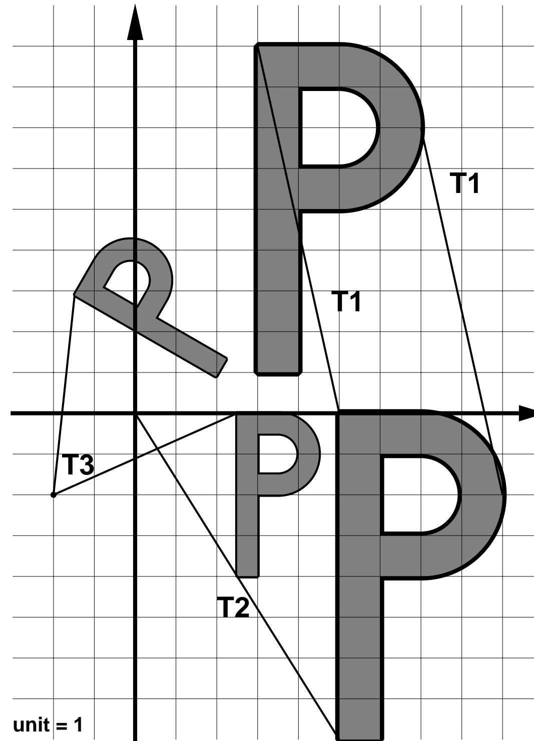


Figure 1.6: Transformations T1,T2 and T3

1.2.5 Frames

The MLgraph system maintains a frame information for sketches and pictures. The frame of an object which contains the object is a rectangle with sides parallel to the axes. It is represented by a value of the following type:

```
type frame = {xmin:float; xmax:float;
              ymin:float; ymax:float};;
```

Frame information is obtained through the following functions:

```
picture_frame : picture -> frame
sketch_frame : sketch -> frame
picture_center : picture -> point
sketch_center : sketch -> point
picture_width : picture -> float
picture_height : picture -> float
sketch_width : sketch -> float
sketch_height : sketch -> float
```

Normally, the frame of an object is the smallest rectangle containing the object. However, when curves are used, the frame includes the control points as well as the curve itself. Also, when transformations are used, the frame of the transformed object is computed using only the frame of the initial object. This leads to frames that are sometimes larger than one would expect. In future versions, tools will be given to compute frames in a more accurate way.

Sometimes, it is useful for the user to determine the frame of an object by himself and under his own responsibility for instance to add blank space around an object. The following functions enable him to do so:

```
force_picture_in_frame : frame -> picture -> picture

type extension = All_ext | Horiz_ext | Vertic_ext | Left_ext
               | Right_ext | Top_ext | Bottom_ext;;


extend_picture_frame : extension -> float -> picture -> picture
extend_sketch_frame : extension -> float -> sketch -> sketch
```

- `force_picture_in_frame fr pict` arbitrarily assigns frame `fr` to picture `pict`.
- `extend_picture_frame ext k pict` extends the frame of picture `pict` using proportion `k` in the direction(s) specified by parameter `ext`.

Some local picture transformations are computed using the frame such as the following ones:

```
rotate_picture : float -> picture -> picture
vflip_picture : picture -> picture
hflip_picture : picture -> picture
scale_picture : float * float -> picture -> picture
```

- `rotate_picture a pict` rotates picture `pict` around its frame center by angle `a`.
- `vflip_picture` and `hflip_picture` perform a symmetry with respect with the vertical and horizontal medians of the frame.
- `scale_picture` scales a picture inside its frame using the left bottom corner as the scaling center.

Figure 1.7 shows the effect of these functions on our favorite example.

An object can also be transformed to fit into a new frame using function:

```
fit_picture_in_frame : picture -> frame -> picture
fit_sketch_in_frame : sketch -> frame -> sketch
```

The figure 1.8 shows the result of fitting a given picture in a given frame.
A simple way to place a picture in a given position is to use the function `center_picture`.

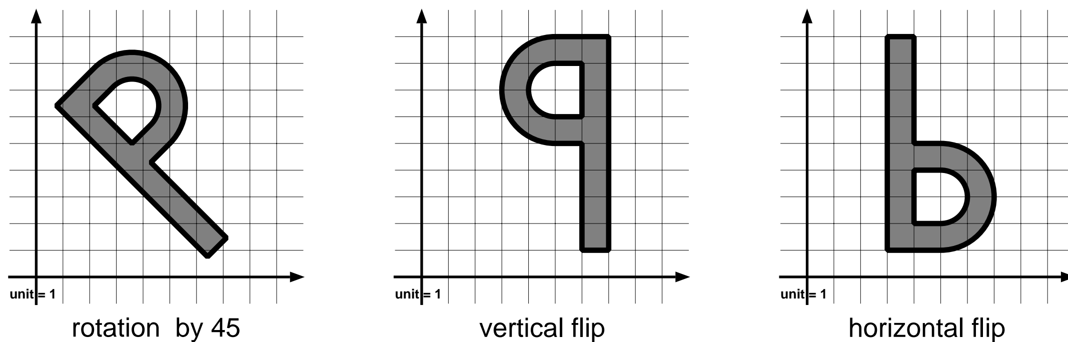


Figure 1.7: Transformations using frames

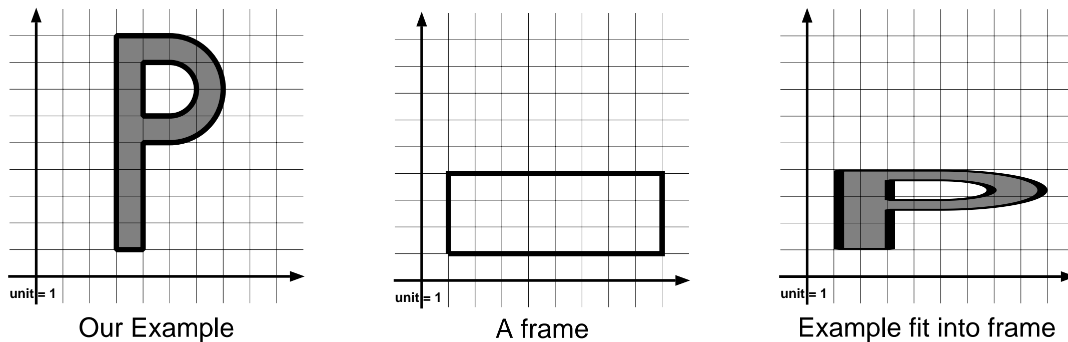


Figure 1.8: Fitting a picture into a given frame

```
center_picture : picture -> point -> picture
```

- `center_picture p pt` translates picture `p` in such a way that its frame center coincides with `pt`.

Let us also mention that it is possible to define a picture that has a frame but nothing in it i.e. a "blank" picture:

```
make_blank_picture : float * float -> picture
```

- `make_blank_picture (h,w)` produces a blank rectangle having height `h` and width `w` and its bottom left corner at the origin. Such pictures can be useful for composition operations defined in section 1.5.

1.3 Text Pictures

Text pictures are strings of characters that are displayed in a given font and with a given color.

Fonts have a style and a size. The corresponding types for these notions are:

```
type font_style =
  Courier  | Courier_Oblique | Courier_Bold | Courier_BoldOblique
| Times_Roman  | Times_Bold  | Times_Italic | Times_BoldItalic
| Helvetica | Helvetica_Bold | Helvetica_Oblique | Helvetica_BoldOblique
| Symbol
| Other_font_style of string;;

type font = { Style : font_style ;  Size : float };;
```

The first 13 font styles are predefined for the PostScript interpretor of the Next computer. The associated frame (bounding box) of each character can change between two PostScript interpretors because the resolution or the drawing are different. The building of new font descriptions is explained in the chapter 3.
Fonts can be defined using function:

```
make_font : font_style -> float -> font
```

The font description is automatically loaded at the first use of a font. If you have some trouble to load fonts, use the `change_graphics_directory` function to indicate the right place of the font descriptions.

The user can ask for the dimensions of any string in any font using functions :

```
text_width : font -> string -> float
text_height : font -> string -> float
text_frame : font -> string -> frame
```

- `text_width` and `text_height` give the width and height of the given string in the given font.
- `text_frame` gives the frame of the given string assuming that the character origin of its first character is at the origin of the coordinate system. This provides all the necessary information to place the text accurately. In particular, it gives a way to compute how much the text spans under and over the reference text line.

Text pictures are defined using functions:

```
make_text_picture : font -> color -> string -> picture
make_textblock_picture : alignment -> float -> font -> color -> string
list -> picture
```

- `make_text_picture ft c s` places the character "origin" of the first character of string `s` at the origin of the coordinate system and draws it with font `ft` and color `c`.
- `make_textblock_picture align sp ft c sl` produces a vertically aligned sequence of text lines `sl` with regular spacing defined by the `sp` parameter and alignment mode defined by the `align` parameter (see subsection 1.5.1 for a definition of type `alignment`).
  The character "origin" of the first character of the first string is placed at the origin.

Fonts are classified into two categories : fixed width (`Courier`) and variable width (`Times`, `Helvetica` or `Symbol`). The figures 1.9 and 1.10 show respectively the characters associated with their frame for the `Courier` font and the `Helvetica_BoldOblique` fonts.

Figure 1.9: A fixed width font : Courier

Figure 1.10: A variable width font : Helvetica-BoldOblique

Fixed width does not mean fixed frame. When using a fixed width font, characters in a string can have different widths but they are placed in such a way that their starting points ares uniformly spaced.

1.4 Bitmaps

A bitmap is basically a bidimensional array of pixels. It has a height, a width and a depth. The depth is the number of bits used for pixels. Possible values are 1,2,4 or 8.

1.4.1 Creation

Bitmaps can be created from scratch and manipulated at the pixel level using the following functions :

```
create_bitmap : int -> int -> int -> bitmap
set_pixel : bitmap -> int -> int -> int -> unit
get_pixel : bitmap -> int -> int -> int
```

- `create_bitmap w h d` creates a bitmap with width `w`, height `h` and depth `d` (1,2,4 or 8 bits). Each pixel has value zero corresponding to the black color.
- `get_pixel b x y` gives the value of the pixel (x,y).

  `set_pixel b x y v` modifies the value of the pixel (x,y) by the new value `v`.

  Pixel values for a bitmap of depth d range from 0 (representing black) to $2^d - 1$ (representing white).

However usually, bitmaps are read from an external string representation.

```
read_bitmap : int -> string -> bitmap
write_bitmap : bitmap -> string -> unit
```

- `read_bitmap d name` reads a bitmap from a file `name` using the integer argument `d` in order to know its depth.
- `write_bitmap name` writes a bitmap to the file `name`.

1.4.2 Modification

Functions are provided to uniformly modify a bitmap or extract a sub_bitmap such as

```
map_bitmap : (int -> int) -> bitmap -> bitmap
convert_bitmap : int * (int -> int) -> bitmap -> bitmap
sub_bitmap : bitmap -> int * int -> int * int -> bitmap
copy_bitmap : bitmap -> bitmap
```

- `map_bitmap` transforms a bitmap into another bitmap having the same width, height and depth by applying a given function to each pixel.
- `convert_bitmap` transforms a bitmap into another bitmap having the same width and height but possibly a new depth given by the first argument.
- `sub_bitmap (x,y) (w,h) b` extracts the sub bitmap of bitmap `b` having left bottom corner at point `(x,y)` and `(w,h)` as width and height.
- `copy_bitmap b` produces a copy of bitmap `b`. This can be useful when using function `set_pixel` which operates destructively.

Figure 1.11 shows an example of a bitmap of depth 1 and its inverted image.

caml_bitmap

map_bitmap (fun 0-> 1| 1->0) caml_bitmap

Figure 1.11: A bitmap transformation

1.4.3 Bitmaps as pictures

To be visualized, a bitmap must be converted into a picture. The following functions perform this translation :

```
make_bitmap_picture : bitmap -> picture
make_bitmap_mask_picture : bitmap -> color -> bool -> picture
```

- `make_bitmap_picture` produces a picture from a gray-level bitmap. All pixels in the bitmap are significant (even “white” pixels will be painted when printing the picture).

- `make_bitmap_mask_picture b c bool` works only for bitmaps with depth `1`. If `bool` is `true`, then only the pixels with value one (white pixels) are significant and painted with bitmap color `c`. If `bool` is `false`, only the pixels with value zero (black pixels) are significant and painted with color `c`.

For example, the figure 1.12 shows the famous `caml_bitmap` visualized over gray rectangles by the following call :`make_bitmap_mask_picture caml_bitmap (Gra 0.0) true`.

Figure 1.12: A camel with its feet wet

1.5 Building complex pictures

It is possible to build complex pictures by grouping more elementary ones using the function `group_pictures`. However, in that case, the user usually has to transform the more basic pictures in order to make them appear in correct relative positions. We now present new grouping functions that incorporate the necessary transformations. Using these composition functions, the users can forget completely about the cartesian plane in which pictures are built. Only the relative sizes of pictures are relevant.

1.5.1 Alignments

Frames can be used to align pictures either horizontally or vertically. Alignment specifications are values of the following type

```
type alignment = Align_Right | Align_Left | Align_Center
               | Align_Top | Align_Bottom;;
```

The alignment functions are:

```
align_horizontally : alignment -> picture list -> picture
align_vertically : alignment -> picture list -> picture
```

Figure 1.13 shows an alignment together with the program to obtain it.

```
(* To obtain a bottom alignment of camels *)

let small = transform_picture (scaling (0.9,0.9));;

let rec iterate f n x =
    if n=0 then []
         else x::iterate f (n-1) (f x);;

align_horizontally Align_Bottom (iterate small 12 camel)
```

Figure 1.13: An alignment

1.5.2 Alignments with scaling

The following functions also perform horizontal or vertical alignments but moreover, they also scale pictures in such a way that their frames fit nicely together either horizontally or vertically.

```
compose_horizontally : picture list -> picture
compose_vertically : picture list -> picture
```

Theses functions also have prefix binary variants:

```
prefix besides : picture -> picture -> picture
prefix over : picture -> picture -> picture
```

Figure 1.14 shows on the upper row three pictures P1, P2, P3 with different vertical sizes and at the bottom the picture compose_horizontally [P1;P2;P3].

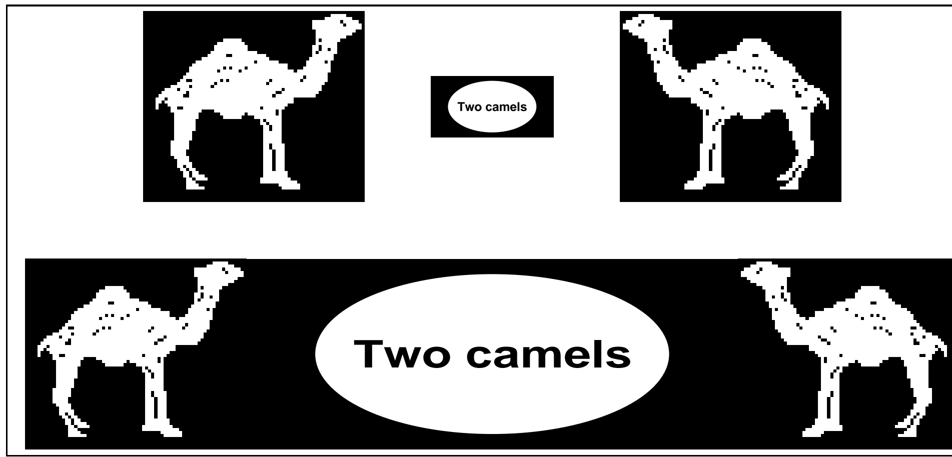


Figure 1.14: Horizontal Composition

1.5.3 More complex picture compositions

Composing pictures horizontally and vertically is not enough and it is often useful to "attach" pictures together in more complex ways. In order to achieve this, pictures can be given handles. A handle is an oriented segment defined by two points. The most simple case is when pictures have one input handle and one output handle but more generally, pictures can also have sets of named handles in input and output. Sets of handles are represented by the following type:

```
type interface = No_handle
               | One_handle of point * point
               | Handles of (string * (point*point)) list;;
```

Pictures have an input interface and an output interface that can be accessed and modified using functions:

```
picture_input_interface : picture -> interface
picture_output_interface : picture -> interface
set_picture_interfaces : picture -> interface * interface -> picture
```

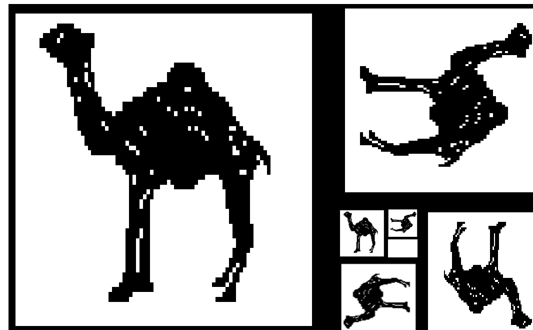
By default, both input and output interfaces are set to "No_handle". The following functions can be used to combine pictures with respect to specified handles:

```
attach_pictures : picture * picture -> picture
named_attach_pictures : picture * picture -> string * string -> picture
```

- `attach_pictures (p1,p2)` combines pictures `p1` and `p2` by transforming `p2` in such a way that its (unique) input handle coincides with the (unique) output handle of `p1`. The input and output handles of the result are the input handle of `p1` and the (transformed) output handle of `p2`.

- `named_attach_pictures (p1,p2) (a1,a2)` combines pictures `p1` and `p2` by transforming `p2` in such a way that its input handle `a2` coincides with output handle `a1` of `p1`. The input handles of the result are the input handles of `p1` augmented with the (transformed) input handles of `p2` except `a2`. The output handles of the result are the output handles of `p1` except `a1` augmented with the (transformed) output handles of `p2` .

Figure 1.15 demonstrates the use of handles. We start with a square bitmap with each side equal to one. It has two handles. The input handle is the segment ((0,0),(1,0)) at the bottom of the picture. The output handle is the segment ((1,1),(1,2-ϕ)) where ϕ is the golden number :$(1+\sqrt{5})/2$. The result is obtained by applying to it the function `gold_spiral` defined in the figure.



```
(* phi is the gold number
   caml_pict is the caml bitmap on the square unit *)

let caml_picture =
  set_picture_interfaces caml_pict
    (One_handle ({xc=0.;yc=0.},{xc=1.;yc=0.}),
     One_handle ({xc=1.;yc=1.},{xc=1.;yc=2.-phi}));;

let rec gold_spiral  =
  function 0 -> caml_picture
      |    n -> attach_pictures (gold_spiral(n-1))
                                 caml_picture;;
```

Figure 1.15: Golden camels

1.6 Using defaults

Giving all the information required to defined pictures is sometimes boring. For instance, after having defined a sketch, the user may want to visualize it without having to bother about a linestyle. In order to simplify things for him, functions using default information have been defined:

```
make_default_draw_picture : sketch -> picture
make_default_closed_draw_picture : sketch -> picture
make_default_fill_picture : sketch -> picture
make_default_text_picture : string -> picture
```

Some other functions use default information such as:

```
add_frame : picture -> picture
```

which makes the frame of a picture visible.

Default information can be accessed and modified using the following functions:

```
default_linewidthcoef : unit -> float
default_linecap : unit -> linecap
default_linejoin : unit -> linejoin
default_miterlimit : unit -> float
default_dashpattern : unit -> int list
default_color : unit -> color
default_fillstyle : unit -> fillstyle
default_font : unit -> font
set_default_linewidthcoef : float -> unit
set_default_linecap : linecap -> unit
set_default_linejoin : linejoin -> unit
set_default_miterlimit : float -> unit
set_default_dashpattern : int list -> unit
set_default_color : color -> unit
set_default_fillstyle : fillstyle -> unit
set_default_font : font -> unit
```

All the default values mentioned have a straightforward meaning except for `linewidthcoef` and `miterlimit`.
`linewidthcoef` is a coefficient that will be multiplied by the average of the height and width of a given picture to determine the linewidth to be used.
`miterlimit` is a PostScript notion which is used to limit the extent of miter joins.

1.7 Producing PostScript Files

The following functions produce a PostScript level 1 file for a picture.

```
ps_file : picture -> string -> unit
eps_file : picture -> string -> unit
```

Function `ps_file` produces plain PostScript whereas function `eps_file` produces Encapsulated PostScript. Encapsulated PostScript includes information about the frame (bounding box) of the picture. This is the format to use when pictures are to be included in LaTeX documents [8] or to be visualized using an interpretor understanding the Encapsulated PostScript format (e.g. Preview on NeXT machines). The main advantage of using Encapsulated PostScript is that there is no need to worry about the picture position on the cartesian plane.
To include pictures in LaTeX documents, use the `epsf` macro described in appendix 3.4.
Plain PostScript is the format to use when the picture is to be sent directly to a printer. In that case, only the part of the picture which is included in the visible region (a rectangle with left bottom corner at the origin, height = 846 and width = 596 for the A4 format) will be printed. When using plain PostScript translation of pictures, the user has normally to be aware of its picture position. However, using functions `scale_picture` and `center_picture`, it is very easy to position correctly a picture in the visible region.

Chapter 2

Extended Examples

We describe two extended examples to show more complex programs and some direct applications of the library. You can find others applications of this library in the following papers [5], [7] and [3].

2.1 Drawing Binary Trees

Drawing binary trees in a pleasant way requires some computation. The main constraint to satisfy is that subtrees should not overlap. Another constraint, almost equally important, is that space should be rather uniformly occupied i.e. given two subtrees with the same father, the respective space to dedicate to each of them depends on their size and shape. Figure 2.1 shows two rather different kinds of binary trees.

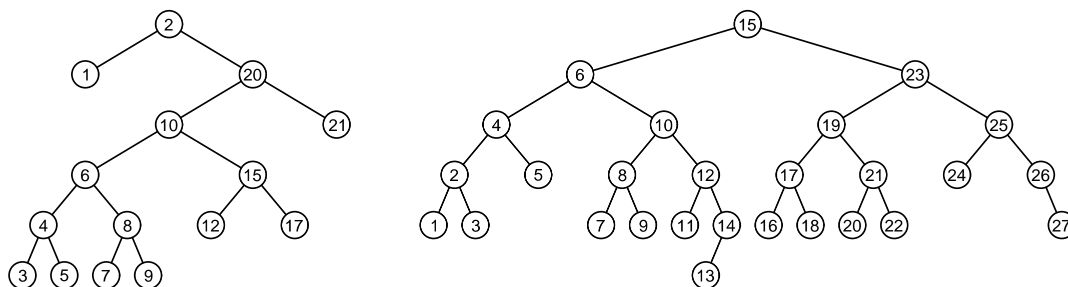


Figure 2.1: Two examples of binary trees

The design principles we have adopted for drawing binary trees are the following :

- At each level of a binary tree, the distance between brother nodes should be constant.
- The distance between brother nodes at level (n+1) should at most equal to the distance at level n.
- Two brother subtrees should be drawn is such a way that at each level, the distance between the rightmost node of the left subtree and the leftmost node of

the right subtree should be at least equal to the standard distance between two brother nodes at that level.

These constraints are taken into account by a function `compute_coef_list` which computes for each binary tree a list of coefficients which indicates the ratio that should be adopted between the distance between two brother nodes at level (n+1) and the distance between two brother nodes at level n.

The function `compute_coef_list` recursively computes for each subtree an information which has shape `(cl,trl)`

- `cl` is the list of reduction coefficients to be applied at each level. It is the information which will finally by used by function `draw_btree` (see below).
- `trl` is a list of triples `(l,r,c)` where
  - `l` is the horizontal distance between tree root and leftmost node at the given level
  - `r` is the horizontal distance between tree root and rightmost node at the given level
  - `c` is the ratio between distance between brother nodes at the given level and distance between brother nodes at level 1

For a given binary tree `t= Node(t1,t2)` the function `compute_coef_list` first computes `(cl1,trl1)` and `(cl2,trl2)` for `t1` and `t2`. Then `cl1` and `cl2` are combined by taking the minimum coefficient at each level giving a new list `cl`. Then, using this new coefficient list, `trl1` and `trl2` are recomputed by function `recompute_triples` giving `trl1'` and `trl2'`. Then, the function `compute_head_coef` computes for each level what should be the reduction coefficient to be applied at the root of tree in order to have the rightmost node of `t1` and the leftmost node of `t2` be separated by distance `c` and takes the minimum of all these coefficients. The method is the following: if `t1` and `t2` were drawn using `cl1` and `cl2`, then the distance between their roots should be at least `r1-l2+c` for `t1` and `t2` to behave nicely at the given level. Therefore the root coefficient should be `1/(r1-l2+c)`

Here are the functions:

```
let recompute_triples cl =  recomp (hd cl,tl cl)
where rec recomp (n,cl) =
function    []      -> []
  | ((l,r,c)::ll) -> (l*.n/.c,r*.n/.c,n):: recomp (n*.(hd cl),tl cl) ll;;

let compute_head_coef (trl1,trl2) =
  it_list min 1.0 (comp_coef (trl1,trl2))
where rec comp_coef =
function ([],_)    ->   []
    |    (_,[])    ->   []
    | ((_,r1,c)::ll1,(l2,_,_)::ll2)
          -> (1.0/.(r1-.l2+.c)) :: comp_coef (ll1,ll2);;
```

```
let combine_triples x (trl1,trl2) =
(-.0.5,0.5,1.0)::comb (trl1,trl2)
where rec comb =
function        [],[]    -> []
   | (l1,r1,c)::ll1 , [] -> (-.0.5+.x*.l1,-.0.5+.x*.r1,c*.x) :: comb(ll1,[])
   | [] , (l2,r2,c)::ll2 -> (0.5+.x*.l2,0.5+.x*.r2,c*.x) :: comb([],ll2)
   | (l1,r1,c)::ll1 , (l2,r2,_)::ll2
        -> (-.0.5+.x*.l1,0.5+.x*.r2,c*.x) :: comb(ll1,ll2);;

let compute_coef_list =   fst o  comp
where rec comp =
function Node {left=Nil;right=Nil;_} -> [1.0],[]
   |  Node {left=t1;right=Nil;_}
         -> let (cl,trl) = comp t1
            in  (1.0::cl,(-.0.5,-.0.5,1.0)
            ::map (fun (l,r,c) -> (-.0.5+.l,-.0.5+.r,c))
                                      trl)
   |  Node {left=Nil;right=t2;_}
         -> let (cl,trl) = comp t2
            in  (1.0::cl,(0.5,0.5,1.0)
            ::map (fun (l,r,c) -> (0.5+.l,0.5+.r,c))
                                      trl)
   |  Node {left=t1;right=t2;_}
         -> let (cl1,trl1) = comp t1
            and (cl2,trl2) = comp t2
            in  let cl = minl(cl1,cl2)
                in let trl1' = recompute_triples cl trl1
                   and trl2' = recompute_triples cl trl2
                   in let x = compute_head_coef (trl1',trl2')
                      in  (1.0::x::tl cl,combine_triples x (trl1',trl2'));;
```

Given this list of coefficients, the drawing of a binary tree becomes straightforward. The drawing function `draw_btree` is a standard recursive function on binary tree which uses the following parameters:

- `drn` is a function for drawing nodes (it is assumed to operates in “fill” mode) *)
- `h` is the height (distance between tree levels)
- `d` is the distance between 2 brother nodes at level 1
- `cl` is a coefficient list
- `pt` is the point where the root should be placed

Function `make_btree_picture` first calls `compute_coef_list` then uses the result is a call to `draw_btree`.

```
let draw_btree drn (h,d,cl,pt) =
```

```
  let LS = {linewidth= h*.0.01;linecap=Buttcap;
            linejoi=Miterjoin;dashpattern=[]}
  in let rec draw_r (d,cl,({xc=x; yc=y} as pt)) =
    function
       Nil -> failwith "Cannot draw an empty tree"
   | Node{info=a;left=Nil;right=Nil}
      -> center_picture (drn a) pt
   | Node{info=a;left=t1;right=t2}
      -> let d=d*.(hd cl)
         in let pt1 = {xc=x-.d/.2.0;yc=y-.h}
            and pt2 = {xc=x+.d/.2.0;yc=y-.h}
            in match (t1,t2) with
                 (_,Nil) -> group_pictures
              [make_draw_picture
            (LS,black)
                                   (make_sketch [Seg [pt;pt1]]);
                              center_picture (drn a) pt;
         draw_r (d,tl cl,pt1) t1]
               | (Nil,_) ->  group_pictures
              [make_draw_picture
            (LS,black)
                                   (make_sketch [Seg [pt;pt2]]);
                              center_picture (drn a) pt;
       draw_r (d,tl cl,pt2) t2]
               |   _     ->  group_pictures
              [make_draw_picture
            (LS,black)
                                   (make_sketch [Seg [pt;pt1]]);
                              make_draw_picture
            (LS,black)
                                   (make_sketch [Seg [pt;pt2]]);
                              center_picture (drn a) pt;
       draw_r (d,tl cl,pt1) t1;
       draw_r (d,tl cl,pt2) t2]


      in draw_r (d,cl,pt)
;;
let make_btree_picture drn (height,d_min,root) t =
  let coef_list = compute_coef_list t
  in let total_coef = it_list mult_float 1.0 coef_list
     in let d= d_min/.total_coef
        in draw_btree drn (height,d,coef_list,root)  t;;
```

Different functions can therefore be used to draw nodes. For instance, if trees are AVL trees, it is possible to indicate in each node whether the subtree corresponding to each node is balanced, or heavier on the left or on the right as shown in figure 2.2.

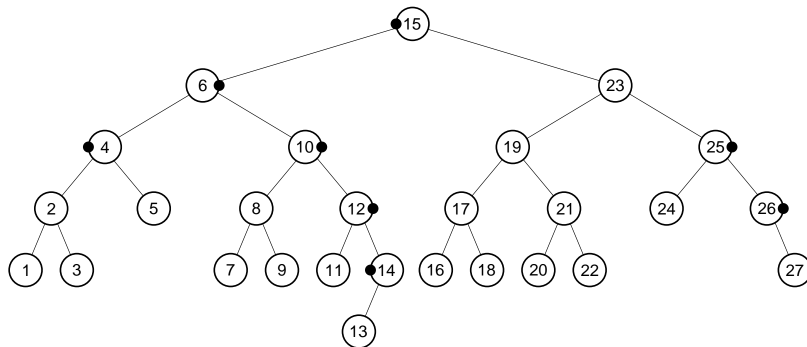


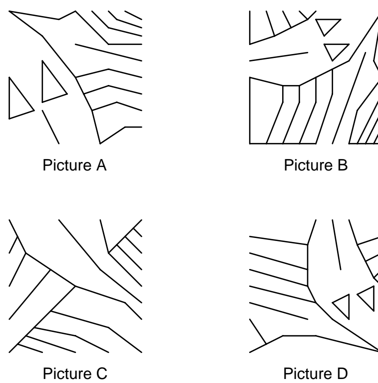
Figure 2.2: An AVL tree

2.2 Escher's Square Limit Picture

Picture composition using alignments is nicely exemplified by the construction of an Escher picture called "Square limit". A programmed version of this picture has been given by Henderson in [6]. A detailed account of Henderson's approach is also given in Course Notes by Cousot [4]. The version presented here uses the same basic pictures but builds the final picture in a different way.

The building blocks of the final picture are four basic pictures `A,B,C,D` and three functions named `trio`, `quartet` and `cycle`.

Here the four basic pictures:

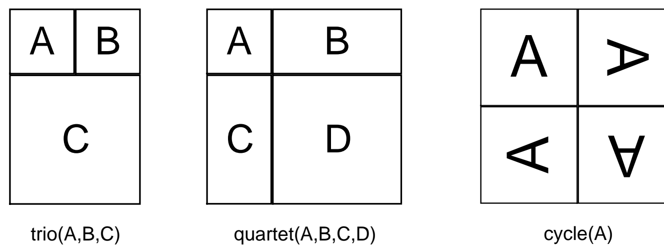


The 4 basic pictures

The three basic functions are defined in the following way:

```
#let rot = rotate_picture 90.0;;
rot : picture -> picture = <fun>
#let trio(p1,p2,p3) =
#   (p1 besides p2) over p3;;
trio : picture * picture * picture -> picture = <fun>
#let quartet (p1,p2,p3,p4) =
#    (p1 besides p2) over (p3 besides p4);;
quartet : picture * picture * picture * picture -> picture = <fun>
#let cycle p =
#    (p besides (rot (rot (rot p))))
#    over
#    ((rot p) besides (rot(rot p)));;
cycle : picture -> picture = <fun>
```

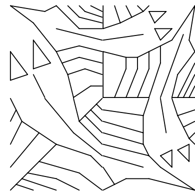
Here is a description of what these functions do:



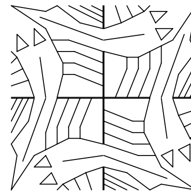
The Caml Light definition of the picture construction is:

```
let small = scale_picture (0.5,0.5);;
let square_limit n (P,Q,R,S) =
   let TT=quartet(P,Q,R,S)
   and UU=cycle (rot Q)
     in
       let step(C,L,T) =
          (quartet(small C
                  ,small(L besides T)
                  ,small((rot T) over (rot L))
                  ,UU)
          ,trio(small L,small T,rot TT)
          ,trio(small L,small T, TT))

       and final_step(C,L,T) =
           quartet(small C,small L, small(rot T),rot Q)
       in

           cycle(final_step(iterate step n (TT,rot TT,UU)));;
```

Here are `TT` and `UU` for basic picture A,B,C,D:

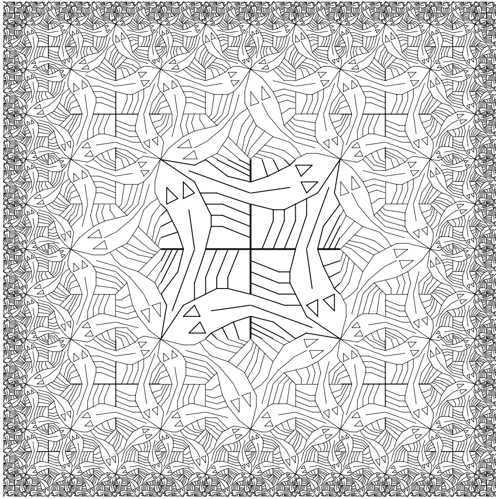


quartet(A,B,C,D)

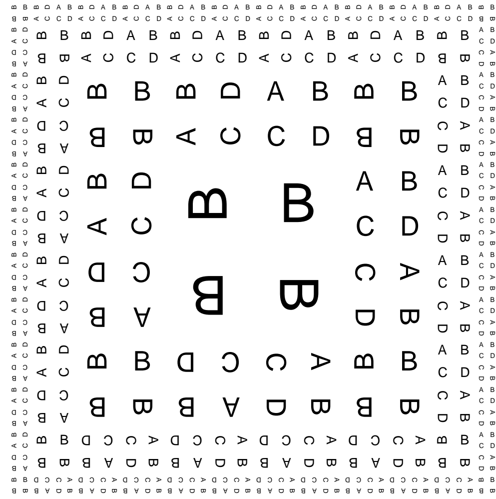


cycle(rot B)

The picture corresponding to the original Escher etching is `square_limit 2 (A,B,C,D)`. It is shown together with its structural description in figure 2.3.



Square Limit



Square Limit structure

Figure 2.3: Final Square Limit

Chapter 3

Installation, Parameterization and Tools

3.1 Installation

The distributed version of the MLgraph library is localized on the ftp server `spi.ens.fr` (*Département d'Informatique et de Mathématique de l'École Normale Supérieure*). Connect yourself onto the `anonymous` account of this computer and go to the `/pub/unix/lang` directory to get the `MLgraph.tar.Z` file. Don't forget to be in binary mode during the transfer because this file is compressed.

The installation is very easy because the distributed package is pre-installed. The compressed file `MLgraph.tar.Z` contains the hierarchical graphics library directory.

To install it :

1) create a new directory, push the `MLgraph.tar.Z` inside and go on :

```
mkdir MYDIR
mv MLgraph.tar.Z MYDIR
cd MYDIR
```

2) uncompress and untar the `MLgraph.tar.Z` file as follow

```
zcat MLgraph.tar.Z | tar -xvf -
```

3) copy the `MLgraph.z?` files and the `MLgraph-lib` directory to `/usr/local/lib/caml-light` directory :

```
cp    MLgraph.z?    /usr/local/lib/caml-light
cp -r MLgraph-lib    /usr/local/lib/caml-light
```

Or if you prefer to select another localization, for example `MYDIR`, don't forget to change the graphics library directory after loading the `MLgraph` code by using the `change_graphics_directory` function in Caml Light.

3.2 Calling MLgraph

You have two ways to use the MLgraph library. The first one is interactive and runs with the Caml Light toplevel. The second one creates an executable file.

3.2.1 interactive session

The library is contained in only one file. This choice facilitates its use.

```
% camllight
>       Caml Light version 0.5

#load_object "MLgraph";;
- : unit = ()
##open "MLgraph";;
#
```

If you have chosen another directory than the usual `/usr/local/lib/caml-light` directory, change the graphics directory as follow :

```
% camllight
>       Caml Light version 0.5

#load_object "./MYDIR/MLgraph";;
- : unit = ()
##open "./MYDIR/MLgraph";;
#change_graphics_directory "./MYDIR";;
- : unit = ()
#
```

3.2.2 independent executable file

If you prefer to work by command line, you need to add the `MLgraph.zo` file to your own code :

```
camlc -o work.exe MLgraph.zo work.ml
```

where `work.ml` is your working file.

Be careful, you are not independent of the `MLgraph-lib` directory which contains font descriptions, several commands and the PostScript headers.

3.3 Using fonts

The predefined font descriptions correspond to the Next computer fonts **Courier**, **Times**, **Helvetica** and **Symbol**. They are correct only for this computer and its printer. For the other PostScript interpretors, and for the other PostScript printers, you need to compute the font description for these 4 fonts and their derived fonts. For that the PostScript program `createfonts.ps`, included in the package in the

`MLgraph-lib/Bin` directory, creates (after uncommenting the last lines) new font descriptions which you can use to replace the predefined ones. Note that all this is important only if you really need to know the exact sizes of characters.

If you want to use other fonts, it is necessary to create them with the `createfonts.ps` program. After that, in your Caml Light program, you need to add the new names of these fonts in the `font_list`. For example, if you want to use the **Ohlfs** font, then :

1) create the font description, called `Ohlfs.font`, and move it to the `MLgraph-lib/Fonts` directory

2) in Caml Light, add the new font as follow

```
#add_font (Other_font_style "Ohlfs",
   {Name="Ohlfs";Height=12.0;Width=12.0;Descr=[||];Descr_bbox=[||]});;
```

to create its entry point.

All font descriptions are automatically loaded at first use.

3.4 Including Pictures inside TEX or LATEX

A new *epsf* TEX macro is given with the MLgraph library (`Headers` directory). It is due to Michel Mauny and Emmanuel Chailloux. TEX performs itself the recognition of the Bounding Box of the included PostScript file. For that, this file must contain in its header a correct Bounding Box.

The macro is `\epsf{filename}` possibly with a specification of the picture size which can have one the following forms :

```
   [xscale=<number>/<number>,yscale=<number>/<number>]
or (hsize=<number>,vsize=<number>)
```

The first form indicates the scale factors by two rational numbers. The second one gives in points the exact size of the drawing. A null `hsize` indicates the hsize scaled by the `vsize` scaling. A null vsize indicates the vsize scaled by the hsize scaling.

For example the figure 2.2 is building as follows :

```
\begin{figure}[hbt]
\begin{center}
\epsf{AVL2.ps}(hsize=300,vsize=0)
\end{center}
\caption{An AVL tree}
\label{avl}
\end{figure}
```

This macro runs for different dvi translators : OzTeX, Dvi2ps and dvips. It is necessary to indicate which is used : `\let\DVITOPS=\dvips` selects the dvips translator.

Bibliography

[1] ADOBE. *PostScript Language : Tutotial and Cookbook*. Addison-Wesley, 1985.

[2] ADOBE. *PostScript Reference Manual*. Addison-Wesley, 1985.

[3] CHAMBERT-LOIR, A., GRANBOULAN, L., AND LEMAIRE, C. Une œuvre d'Escher en CAML. Tech. rep., "École normale supérieure", 1991. Rapport de projet de Magistère.

[4] COUSOT, P. Cours d'Informatique de l'École Polytechnique. Paris, 1988.

[5] CRÉGUT, P. An Abstract Machine for the Normalization of λ-terms. In *Lisp and Functional Programming* (1990), ACM.

[6] HENDERSON, P. Functional Geometry. In *Symposium on Lisp and Functional Programming* (1982), ACM.

[7] LAFONT, Y. Penrose diagrams and 2-dimensional rewriting. In *Symposium on Applications of categories in Computer Science* (1992), Cambridge University Press, LMS Lecture Notes Series.

[8] LAMPORT, L. *LaTeXUser's Guide and Reference Manual*. Addison-Wesley, 1986.

[9] MAUNY, M. Functional Programming using CAML Light. Tech. rep., INRIA, Sept. 1991.

Glossary

List of the MLgraph predefined values

add_frame : picture → picture . . . 19

align_horizontally : alignment → picture list → picture . . . 16

align_vertically : alignment → picture list → picture . . . 16

attach_pictures : picture * picture → picture . . . 17

besides : picture → picture → picture . . . 17

center_picture : picture → point → picture . . . 10

change_graphics_directory : string → unit . . . 12, 30

clip_picture : clipstyle → sketch → picture → picture . . . 7

compose_horizontally : picture list → picture . . . 16

compose_transformations : transformation list → transformation . . . 5

compose_vertically : picture list → picture . . . 16

convert_bitmap : int * (int → int) → bitmap → bitmap . . . 14

copy_bitmap : bitmap → bitmap . . . 14

create_bitmap : int → int → int → bitmap . . . 14

ctrans : transformation → transformation → transformation . . . 5

default_color : unit → color . . . 19

default_dashpattern : unit → int list . . . 19

default_fillstyle : unit → fillstyle . . . 19

default_font : unit → font . . . 19

default_linecap : unit → linecap . . . 19

default_linejoin : unit → linejoin . . . 19

default_linewidthcoef : unit → float . . . 19

default_miterlimit : unit → float . . . 19

eps_file : picture → string → unit . . . 19

extend_picture_frame : extension → float → picture → picture . . . 10

extend_sketch_frame : extension → float → sketch → sketch . . . 10

fit_picture_in_frame : picture → frame → picture 10
fit_sketch_in_frame : sketch → frame → sketch 10
force_picture_in_frame : frame → picture → picture 10
get_pixel : bitmap → int → int → int 14
group_pictures : picture list → picture 8
group_sketches : sketch list → sketch 6
handle_transform : point * point → point * point → transformation 5
hflip_picture : picture → picture 10
id_trans : transformation 5
inverse_transformation : transformation → transformation 5
line_symmetry : point * point → transformation 5
make_bitmap_mask_picture : bitmap → color → bool → picture 15
make_bitmap_picture : bitmap → picture 15
make_blank_picture : float * float → picture 11
make_closed_draw_picture : linestyle * color → sketch → picture 7
make_default_closed_draw_picture : sketch → picture 19
make_default_draw_picture : sketch → picture 19
make_default_fill_picture : sketch → picture 19
make_default_text_picture : string → picture 19
make_draw_picture : linestyle * color → sketch → picture 7
make_fill_picture : fillstyle * color → sketch → picture 7
make_font : font_style → float → font 12
make_sketch : geom_element list → sketch 6
make_textblock_picture : alignment → float → font → color → string list → picture . 12
make_text_picture : font → color → string → picture 12
make_transformation : float * float * float * float * float * float → transformation . 5
map_bitmap : (int → int) → bitmap → bitmap 14
named_attach_pictures : picture * picture → string * string → picture 17
over : picture → picture → picture 17
picture_center : picture → point 9
picture_frame : picture → frame 9
picture_height : picture → float 9
picture_input_interface : picture → interface 17
picture_output_interface : picture → interface 17
picture_width : picture → float 9

point_symmetry : point → transformation 5
ps_file : picture → string → unit 19
read_bitmap : int → string → bitmap 14
rotate_picture : float → picture → picture 10
rotation : point → float → transformation 5
scale_picture : float * float → picture → picture 10
scaling : float * float → transformation 5
set_default_color : color → unit 19
set_default_dashpattern : int list → unit 19
set_default_fillstyle : fillstyle → unit 19
set_default_font : font → unit 19
set_default_linecap : linecap → unit 19
set_default_linejoin : linejoin → unit 19
set_default_linewidthcoef : float → unit 19
set_default_miterlimit : float → unit 19
set_picture_interfaces : picture → interface * interface → picture 17
set_pixel : bitmap → int → int → int → unit 14
sketch_center : sketch → point 9
sketch_frame : sketch → frame 9
sketch_height : sketch → float 9
sketch_width : sketch → float 9
sub_bitmap : bitmap → int * int → int * int → bitmap 14
text_frame : font → string → frame 12
text_height : font → string → float 12
text_width : font → string → float 12
transform_picture : transformation → picture → picture 8
transform_sketch : transformation → sketch → sketch 8
translation : float * float → transformation 5
vflip_picture : picture → picture 10
write_bitmap : bitmap → string → unit 14

Index

alignment 16
alignments with scaling 16
bitmap 13
cartesian plane 3
clipping 7, 8
color 7
defaults 19
epsf 20
fonts 11
frame 9
frame extension 10
frame modifications 10
geometric_element 3
grouping 6, 8
handles 17
interfaces 17
line style 7
line_symmetry 5
picture 7
point 3
point_symmetry 5
postscript files 19
rotation 5
scaling 5
sketch 6
text 11
text block 13
transformation 4
transformation composition 5
translation 5

Contents

1 Description of the System **3**

1.1 Basic Geometric Notions . . . 3

1.1.1 Geometric elements . . . 3

1.1.2 Transformations . . . 4

1.2 Sketches and Pictures . . . 6

1.2.1 Sketches . . . 6

1.2.2 Painting Information . . . 7

1.2.3 Pictures . . . 7

1.2.4 Applying transformations to sketches and pictures . . . 8

1.2.5 Frames . . . 9

1.3 Text Pictures . . . 11

1.4 Bitmaps . . . 13

1.4.1 Creation . . . 14

1.4.2 Modification . . . 14

1.4.3 Bitmaps as pictures . . . 15

1.5 Building complex pictures . . . 16

1.5.1 Alignments . . . 16

1.5.2 Alignments with scaling . . . 16

1.5.3 More complex picture compositions . . . 17

1.6 Using defaults . . . 19

1.7 Producing PostScript Files . . . 19

2 Extended Examples **21**

2.1 Drawing Binary Trees . . . 21

2.2 Escher's Square Limit Picture . . . 25

3 Installation, Parameterization and Tools **29**

3.1 Installation . . . 29

3.2 Calling MLgraph . . . 30

3.2.1 interactive session . . . 30

3.2.2 independent executable file . . . 30

3.3 Using fonts . . . 30

3.4 Including Pictures inside TEX or LATEX . . . 31

Bibliography **33**

List of Figures

1.1 Points in the cartesian plane 3
1.2 Geometric elements 4
1.3 A sketch representing letter P 6
1.4 Two pictures obtained from the same sketch 8
1.5 A superposition of pictures P1 and P2 8
1.6 Transformations T1,T2 and T3 9
1.7 Transformations using frames 11
1.8 Fitting a picture into a given frame 11
1.9 A fixed width font : Courier 13
1.10 A variable width font : Helvetica-BoldOblique 13
1.11 A bitmap transformation 15
1.12 A camel with its feet wet 15
1.13 An alignment 16
1.14 Horizontal Composition 17
1.15 Golden camels 18

2.1 Two examples of binary trees 21
2.2 An AVL tree 25
2.3 Final Square Limit 27