The MLgraph system

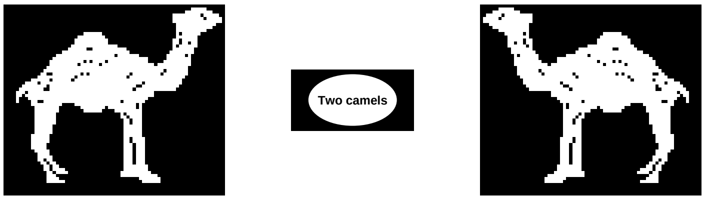
release 2.1

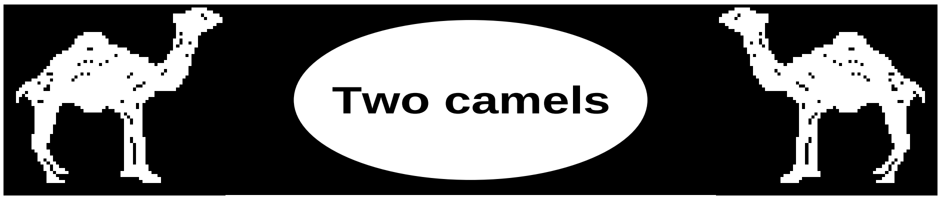
Documentation and user's manual

Emmanuel CHAILLOUX[1] – Guy COUSINEAU[2]

Ascánder SUÁREZ[3]

August 27, 1997





[1]Laboratoire d'Informatique de Paris 6 - 4, place Jussieu - Université Pierre et Marie Curie - 75252 Paris Cédex 05, France. Electronic mail: Emmanuel.Chailloux@litp.ibp.fr

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

[3]Universidad Simón Bolívar - Departamento de Computación - Valle de Sartenejas - Caracas 1080 - Venezuela. E-mail : suarez@usb.ve

Contents

Introduction

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 and Objective Caml.
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. However, the PostScript language is replaced by ML which allows defining images in a natural functional style and therefore to define higher level picture constructions.
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. It is also possible to use non-linear transformations but in that case, these transformations are performed by ML before the translation to PostScript.
This manual is divided into four sections. The first section is a general presentation of MLgraph. The second section details two extended examples. The third section describes mlPicTEX, an MLgraph extension for the design of figures which might contain text composed with LaTeX. The fourth 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, an index of MLgraph notions, a glossary of mlPicTEX functions an an index of mlPicTEX options.

Acknowledgments

We would like to thanks Michel Mauny for letting us use his Camlp program (Caml prettyfier and interface for LaTeX), and Roberto Dicosmo for his help on LaTeX macros. We are grateful to Pierre Crégut made many usefull suggestions.

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).

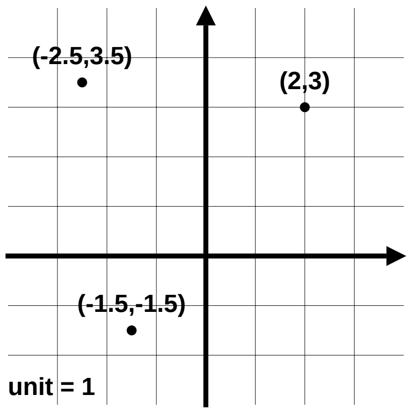


Figure 1.1: Points in the cartesian plane

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 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.

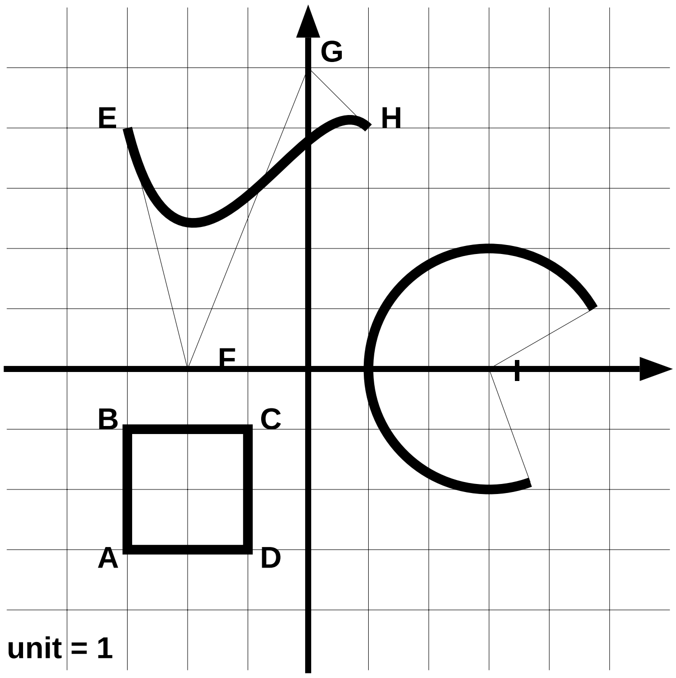


Figure 1.2: Geometric elements

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).

A transformation can be applied to a point using transform_point : transformation $\rightarrow$ point $\rightarrow$ point

The library offers various functions to build transformations:

id_trans : transformation
translation : float $*$ float $\rightarrow$ transformation
rotation : point $\rightarrow$ float $\rightarrow$ transformation
scaling : float $*$ float $\rightarrow$ transformation
line_symmetry : point $*$ point $\rightarrow$ transformation
hsymmetry : float $\rightarrow$ transformation
vsymmetry : float $\rightarrow$ transformation
point_symmetry : point $\rightarrow$ transformation
handle_transform : point $*$ point $\rightarrow$ point $*$ point $\rightarrow$ transformation
make_transformation : float $*$ float $*$ float $*$ float $*$ float $*$ float
$\rightarrow$ 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 (p_1,p_2) builds a symmetry according to the line defined by the two points p_1 and p_2. It fails if the two points are equal.
- hsymmetry a builds a symmetry according to the line defined by the equation y=a.
- vsymmetry a builds a symmetry according to the line defined by the equation x=a.
- point_symmetry p builds a symmetry according to point p.
- handle_transform (pt_1,pt_2) (pt_3,pt_4) builds a linear transformation that maps pt_1 to pt_3 and pt_2 to pt_4. It fails if $\mathsf{pt}_1 = \mathsf{pt}_2$ or $\mathsf{pt}_3 = \mathsf{pt}_4$. 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
inverse_transformation : transformation → transformation

- compose_transformations tr_list produces the composition of transformation list tr_list, computed from right to left.
- inverse_transformation tr produces the inverse of transformation tr provided 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 and convex hull information (cf. 1.2.5) or 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.
Also, the function:

ungroup_sketch : sketch → sketch list

is the inverse of group_sketches.
Figure 1.3 shows a text defining a sketch together with its graphical representation.

```
let skp=
group_sketches
 [(make_sketch
   [Arc({xc=5.;yc=7.},2.,-90.,90.);
      Seg [{xc=5.;yc=9.};{xc=3.;yc=9.};
           {xc=3.;yc=1.};{xc=4.;yc=1.};
           {xc=4.;yc=5.};{xc=5.;yc=5.}]])];
  (make_sketch
   [Arc({xc=5.;yc=7.},1.,-90.,90.);
      Seg [{xc=5.;yc=8.};{xc=4.;yc=8.};
           {xc=4.;yc=6.};{xc=5.;yc=6.}]])]
```

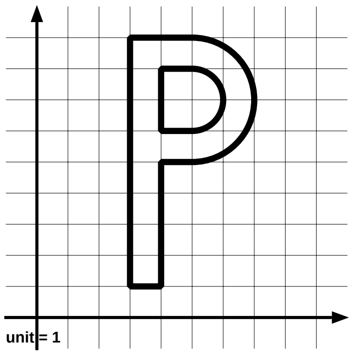


Figure 1.3: A sketch representing letter P

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 types described below.

Line endings

#type linecap = Buttcap | Squarecap | Roundcap;;

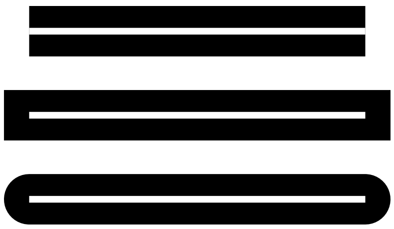


Figure 1.4: Buttcap, Squarecap and Roundcap

Figure 1.4 shows the line endings correspondings to the three constructors. The white lines inside the black lines show where are the exact ends of the line. With Buttcap, the line is squared off at the end points. With Squarecap and Roundcap, the line goes beyond the end points at a distance equal to half the line width.

Line joins

#type linejoin = Beveljoin | Roundjoin | Miterjoin;;

Figure 1.5: Beveljoin, Roundjoin and Miterjoin

Figure 1.5 shows the line joins correspondings to the three constructors. With Beveljoin, the two lines are ended as with Buttcap and the resulting notch is filled. With Roudjoin, a circular arc with diameter equal to the line width and center at the meetig point is filled, producing a rounded corner. With Miterjoin, the lines are extended until they meet at an angle.

Line styles

```
#type linestyle = {linewidth:float;
                   linecap:linecap;
                   linejoin:linejoin;
                   dashpattern:int list};;
```

A line style is defined by a line width, a line cap, a line join and a dash pattern which is an integer list. If the list is empty, then the line is a plain stroke. If the list contains one integer n, then the line is draw by alternating a stroke of length n followed by a blank of length n. If the list contains two integers m and n, then the line is draw by alternating a stroke of length m followed by a blank of length n.

Fill styles and clip styles

```
#type fillstyle = Nzfill | Eofill;;
```

```
#type clipstyle = Nzclip | Eoclip;;
```

These values are used to determine what is the interior of a sketch for filling or clipping operations. For simple sketch, it is intuitively clear what is the interior but it is not when the sketch segments cross each others. When Nzfill or Nzclip are used, a point is consider to be part of the interior according to the non-zero winding number. A ray is (conceptually) drawn from the point to infinity. Starting with the value 0, we add 1 when the ray crosses a sketch segment going from left to right and subtract 1 when when the ray crosses a sketch segment going from right to left. If the final count is zero, the point is outside otherwise it is inside. When Eofill or Eoclip are used, a point is considered to be part of the interior according to the even-odd rule. A ray is (conceptually) drawn from the point to infinity. Starting with the value 0, we add 1 when the ray crosses a sketch segment. If the final count is even, the point is outside otherwise it is inside.

Colors

#type color = Rgb **of** float $*$ float $*$ float
| Hsb **of** float $*$ float $*$ float
| Gra **of** float;;

A color can be defined in the Red-Green-Blue model, in the Hue-Saturation-Brightness model or as a gray level. All numbers used in the definition of a color are floating point numbers between 0 and 1.
In the Red-Green-Blue model (constructor Rgb), the three parameters tell how much of red, green and blue components should be used to produces the color.
In the Hue-Saturation-Brightness model (constructor Hsb), colors are arranged in a conventional color circle where the angle determines a color and the distance to the center determines a saturation in relation with the brightness. Maximal saturation is 1 and 0 corresponds to a gray level. The brightness is a separate parameter where 0 corresponds to black and 1 to maximal intensity.
To determine a color properly in any of these two models, one should use a color palette on its computer.
The gray level (constructor Gra) ranges from 0 (black) to 1 (white).
Some variables contain predefined colors.

black : color
white : color
red : color
green : color
blue : color
cyan : color
magenta : color
yellow : color

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 $\rightarrow$ sketch $\rightarrow$ picture
make_closed_draw_picture : linestyle $*$ color $\rightarrow$ sketch $\rightarrow$ picture
make_fill_picture : fillstyle $*$ color $\rightarrow$ sketch $\rightarrow$ 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 1.6.

```
(* To obtain a draw picture *)
let P1=
  let lsty = {linewidth=0.2;linecap=Buttcap;
             linejoin=Beveljoin;dashpattern=[]}
  in  make_closed_draw_picture (lsty,Gra 0.0) skp;;

(* To obtain of fill picture *)
let P2=
   make_fill_picture (Eofill,Gra 0.5) skp;;
```

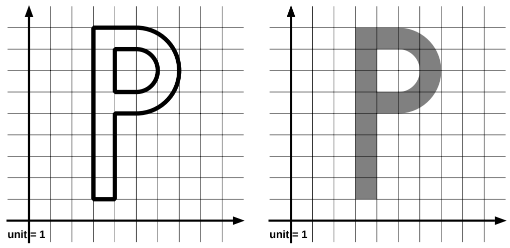


Figure 1.6: Two pictures obtained from the same sketch

It is also possible to use a sketch to "clip" part of a picture.

clip_picture : clipstyle $\rightarrow$ sketch $\rightarrow$ picture $\rightarrow$ 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 or ungrouped using the functions :

group_pictures : picture list $\rightarrow$ picture
ungroup_picture : picture $\rightarrow$ picture list

The grouping is performed from left to right. Each new picture can cover previous ones partially or totally. Figure 1.7 shows what is obtained by grouping the two pictures of figure 1.6.

1.2.4 Applying transformations to sketches and pictures

Transformations are applied to sketches and pictures using functions

```
(* To superpose P1 and P2 *)

let P3=
  group_pictures [P2;P1]
```

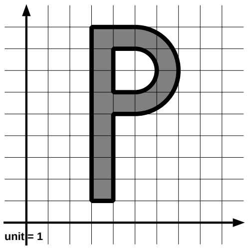


Figure 1.7: A superposition of pictures P1 and P2

transform_sketch : transformation $\rightarrow$ sketch $\rightarrow$ sketch
transform_picture : transformation $\rightarrow$ picture $\rightarrow$ picture

Figure 1.8 describes the effect of applying successively transformations $\mathsf{T}_1=$ translation $(2.,-9.)$,
$\mathsf{T}_2=$ scaling $(0.5,0.5)$ and $\mathsf{T}_3=$ rotation $\{\mathsf{xc}=-2.;\mathsf{yc}=-2.\}$ 60.0 to our basic picture.
The following functions allow transforming a sketch or a picture without having first to define a transformation.

translate_sketch : float $*$ float $\rightarrow$ sketch $\rightarrow$ sketch
translate_picture : float $*$ float $\rightarrow$ picture $\rightarrow$ picture
scale_sketch : float $*$ float $\rightarrow$ sketch $\rightarrow$ sketch
scale_picture : float $*$ float $\rightarrow$ picture $\rightarrow$ picture
rotate_sketch : float $\rightarrow$ sketch $\rightarrow$ sketch
rotate_picture : float $\rightarrow$ picture $\rightarrow$ picture
vflip_sketch : sketch $\rightarrow$ sketch
vflip_picture : picture $\rightarrow$ picture
hflip_sketch : sketch $\rightarrow$ sketch
hflip_picture : picture $\rightarrow$ picture

For pictures, these functions also have an abbreviated name:

translate : float $*$ float $\rightarrow$ picture $\rightarrow$ picture
scale : float $*$ float $\rightarrow$ picture $\rightarrow$ picture
rotate : float $\rightarrow$ picture $\rightarrow$ picture
vflip : picture $\rightarrow$ picture
hflip : picture $\rightarrow$ picture

1.2.5 Frames and convex hulls

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:

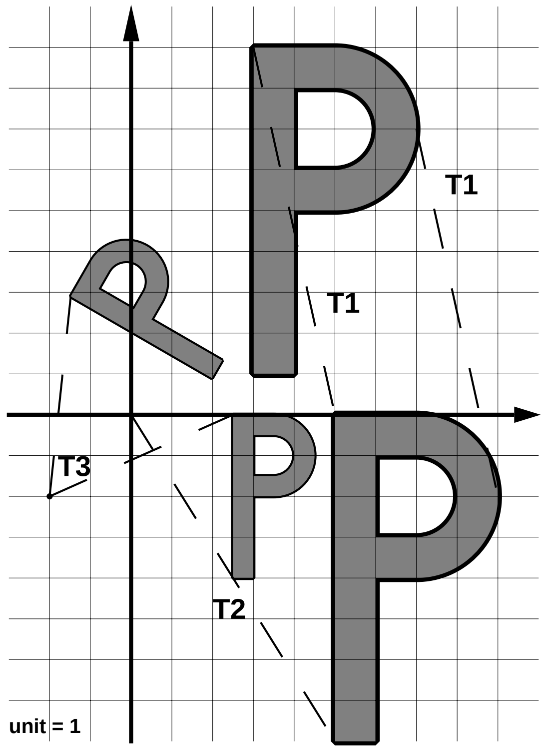


Figure 1.8: Transformations T1,T2 and T3

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

Obtaining frame information

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

The center of a sketch or a picture is the center of its frame.

How frame information is computed

When only translations and scalings are used, the frame of the resulting object can be computed from the frame of the object that is translated or scaled. But when rotations are used, this is not possible. Therefore, MLgraph also

provide convex hulls for sketches and pictures. For any linear transformation, the convex hull of the resulting object can be computed from the convex hull of the object being transformed. If the frames are then computed from convex hulls, they will always be exact.
However, convex hull computations can be expensive. So, MLgraph offers two different modes. In exact frame mode, convex hulls and frames are computed for any sketch and picture. In the other mode, which is the default, only frames are computed and can become incorrect when rotations are used. The user can switch between the two modes or obtain the current mode using the functions:

set_exact_frame_mode : bool → unit
get_exact_frame_mode : unit → bool

The value true correspond to exact frame mode. When using in exact frame mode a sketch or a picture that has been constructed in the other mode, the convex hulls and frames are recomputed automatically.
Even when not in exact frame mode, the user can ask MLgraph to compute exact frame information using the functions:

recompute_sketch_hull : sketch → sketch
recompute_picture_hull : picture → picture

The convex hull and frame information associated to the sketch or picture are physically updated.
The user can ask wheter a sketch or a picture have exact frame information using the functions:

has_exact_frame_sketch : sketch → bool
has_exact_frame_picture : picture → bool
has_exact_frame : picture → bool

The user can also ask for the convex hull of a sketch or a picture:

sketch_hull : sketch → point list
picture_hull : picture → point list

However, if the argument does not have exact frame information these functions will fail. Therefore, the user should test first if the object has exact frame and ask MLgraph to recompute hull if necessary.

Setting accuracy

In exact frame mode, the convex hull computations require a discretization of arcs and curves which are replaced by point sequences. In this process, there is a trade-off between accuracy and efficiency. The user can decide how many points should be used for a circle or a curve using the functions:

set_discrete_circle_number : int → unit
get_discrete_circle_number : unit → int
set_discrete_curve_number : int → unit
get_discrete_curve_number : unit → int

Visualizing frames and convex hulls

A frame or convex hull can be transformed into a picture:

make_frame_picture : linestyle $*$ color $\rightarrow$ frame $\rightarrow$ picture
make_hull_picture : linestyle $*$ color $\rightarrow$ point list $\rightarrow$ picture

The frame or convex hull of an picture can be transformed into a picture:

get_picture_frame_as_picture : picture $\rightarrow$ picture
get_picture_hull_as_picture : picture $\rightarrow$ picture

Modifying frame information

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. This is possible only in non exact frame mode.
A blank rectangle with given height and width or a blank square with given edge size can be obtained with the functions:

blank_rectangle : float $*$ float $\rightarrow$ picture
blank_square : float $\rightarrow$ picture

The following functions change frames of sketches or pictures:

extend_sketch_frame : extension $\rightarrow$ float $\rightarrow$ sketch $\rightarrow$ sketch
extend_picture_frame : extension $\rightarrow$ float $\rightarrow$ picture $\rightarrow$ picture
force_picture_in_frame : frame $\rightarrow$ picture $\rightarrow$ picture

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

Extensions are values of the following type:

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

Transformations relative to frames

Many transformation functions use the frame of their argument:

- 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 and center the result on the frame center of the argument.

Fitting an object into a frame

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

fit_picture_in_frame : picture → frame → picture
fit_sketch_in_frame : sketch → frame → sketch

The figure 1.9 shows the result of fitting a given picture in a given frame.

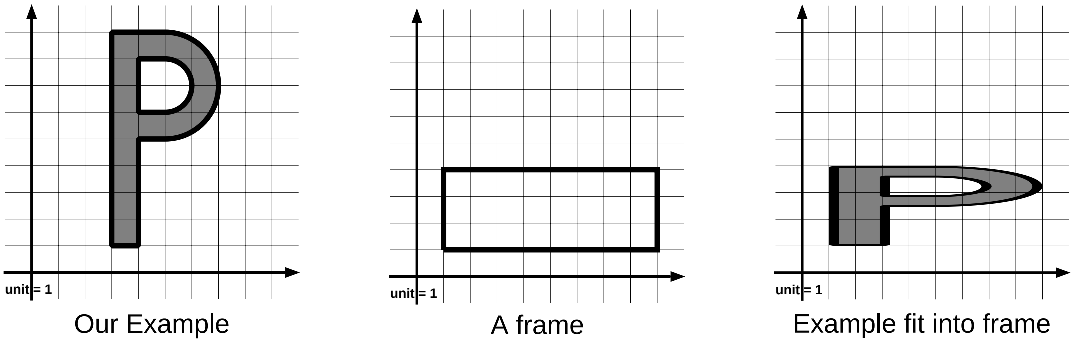


Figure 1.9: Fitting a picture into a given frame

Placing objects

A simple way to place a picture in a given position is to use the function center_picture.

center_picture : picture → point → picture

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

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 = { font_style : font_style ; font_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 4.
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 1.10 and 1.11 show respectively the characters associated with their frame for the Courier font and the Helvetica_BoldOblique fonts.

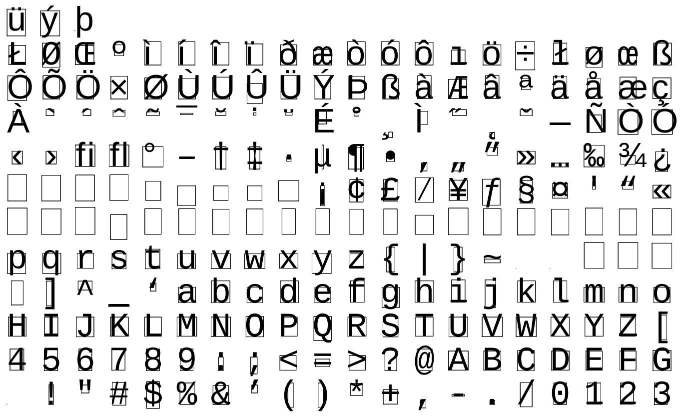


Figure 1.10: A fixed width font : Courier

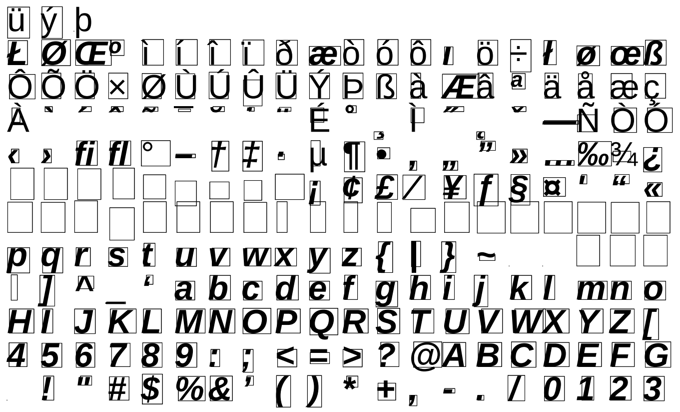


Figure 1.11: A variable width font : Helvetica-BoldOblique

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 $\rightarrow$ int $\rightarrow$ int $\rightarrow$ bitmap
set_pixel : bitmap $\rightarrow$ int $\rightarrow$ int $\rightarrow$ int $\rightarrow$ unit
get_pixel : bitmap $\rightarrow$ int $\rightarrow$ int $\rightarrow$ 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 $\rightarrow$ string $\rightarrow$ bitmap
write_bitmap : bitmap $\rightarrow$ string $\rightarrow$ 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 $\rightarrow$ int) $\rightarrow$ bitmap $\rightarrow$ bitmap
convert_bitmap : int $*$ (int $\rightarrow$ int) $\rightarrow$ bitmap $\rightarrow$ bitmap
sub_bitmap : bitmap $\rightarrow$ int $*$ int $\rightarrow$ int $*$ int $\rightarrow$ bitmap
copy_bitmap : bitmap $\rightarrow$ 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 1.12 shows an example of a bitmap of depth 1 and its inverted image.

Figure 1.12: 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 1.13 shows the famous caml_bitmap visualized over gray rectangles by the following call : make_bitmap_mask_picture caml_bitmap (Gra 0.0) true.

Figure 1.13: 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 coordinates. Only the relative sizes of pictures are relevant.

1.5.1 Alignments

Pictures can be aligned 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 1.14 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 1.14: 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 1.15 shows on the upper row three pictures
p_1, p_2, p_3 with different vertical sizes and at the bottom the picture compose_horizontally [p_1;p_2;p_3].

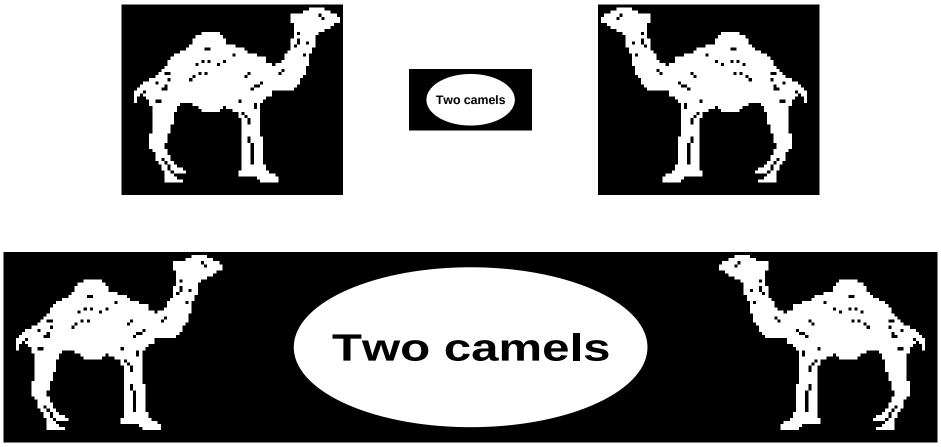


Figure 1.15: Horizontal Composition

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 $\rightarrow$ picture
named_attach_pictures : picture $*$ picture $\rightarrow$ string $*$ string $\rightarrow$ picture

- attach_pictures (p_1,p_2) combines pictures p_1 and p_2 by transforming p_2 in such a way that its (unique) input handle coincides with the (unique) output handle of p_1. The input and output handles of the result are the input handle of p_1 and the (transformed) output handle of p_2.
- named_attach_pictures (p_1,p_2) (a_1,a_2) combines pictures p_1 and p_2 by transforming p_2 in such a way that its input handle a_2 coincides with output handle a_1 of p_1. The input handles of the result are the input handles of p_1 augmented with the (transformed) input handles of p_2 except

  a_2. The output handles of the result are the output handles of p_1 except a_1 augmented with the (transformed) output handles of p_2 .

Figure 1.16 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.

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:

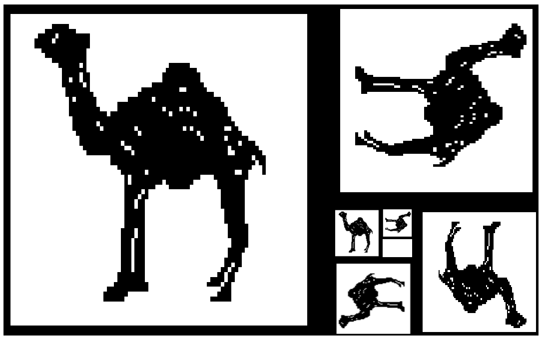
make_default_draw_picture : sketch $\rightarrow$ picture
make_default_closed_draw_picture : sketch $\rightarrow$ picture
make_default_fill_picture : sketch $\rightarrow$ picture
make_default_text_picture : string $\rightarrow$ picture

Some other functions use default information such as:

add_frame : picture $\rightarrow$ picture

which makes the frame of a picture visible.
Default information can be accessed and modified using the following functions:

default_linewidthcoef : unit $\rightarrow$ float
default_linecap : unit $\rightarrow$ linecap
default_linejoin : unit $\rightarrow$ linejoin
default_miterlimit : unit $\rightarrow$ float



```
(* 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 1.16: Golden camels

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, and 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 $\rightarrow$ string $\rightarrow$ unit
eps_file : picture $\rightarrow$ string $\rightarrow$ 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 [6] 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 chapter 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.
The bounding boxes for the PostScript translation of a picture are computed using the frame information. However, the frame information does not include line widthes. Therefore, a slight extension of the frame is performed so that the PostScript image includes the MLgraph image. However, il you use large line width and sharp angles with miter joins, this might not be enough. In that case, you shoud increase the frame extension coefficient which is equal to 0.02 by default.

set_frame_extension_coef : float $\rightarrow$ unit

Chapter 2

Extended Examples

We describe two extended examples: displaying binary trees and making tilings. In these examples, complex computations are necessary to correctly place the objects but these computations are outside the scope of this manual. Therefore, we only describe how pictures are built once these computations have been performed. Interrested readers will find more information in [4].

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.

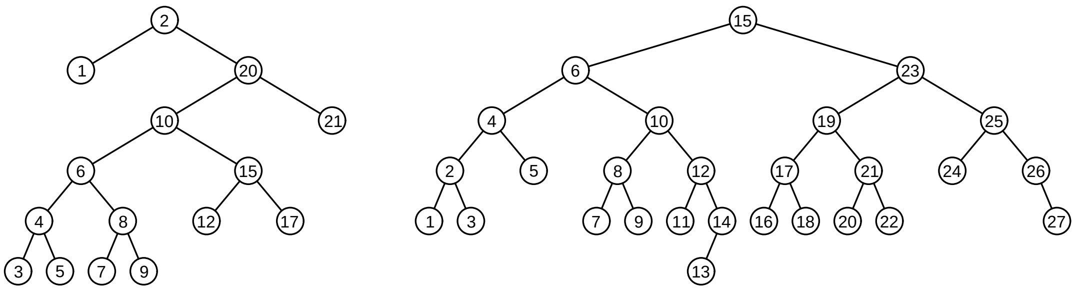


Figure 2.1: Two examples of binary trees

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. This function is not described here.
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;
            linejoin=Miterjoin;dashpattern=[ ]}
  in let rec draw_r (d,cl,({xc=x; yc=y} as pt)) =
      function
       Nil → center_picture (make_blank_picture (0.,0.)) pt
      | 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;
                            drawr (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;
                      drawr (d,tl cl,pt1) t1;
                      drawr (d,tl cl,pt2) t2]
    in drawr (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.

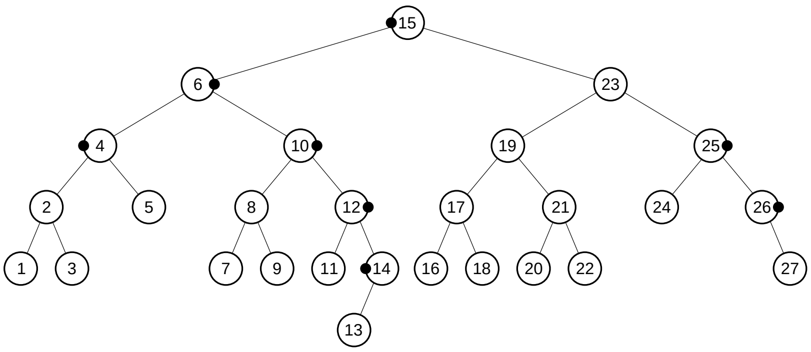


Figure 2.2: An AVL tree

2.2 Tilings

2.2.1 Principles of tiling constructions

We use the word “tiling” here to mean filling a surface with non-overlapping copies of a single pattern. Tilings can be found is everyday life: street, kitchen,

bathroom pavements are examples.
Scientists became interrested in tilings by studying cristals. Cristallographs have classified plane and space tilings by their symmetry groups and shown for instance that in the euclidian plane, there exists only 17 possible symmetry groups of periodic tilings. Since then, the study of tilings has remained a subject of high interest for physicist and mathematicians [5].
Tilings have also often been used to decorate monuments, most noticeably in the arab civilization. In this century, the dutch artist M.C. Escher has produced many tilings based on animal or human shapes. Here, our examples will be based on an animal shape: the kangaroo [1]. Such a kangaroo appears in figure 2.3.

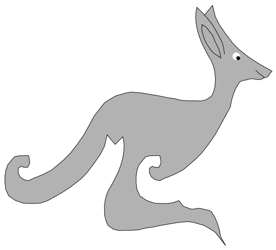


Figure 2.3: A kangaroo

A tiling is defined by a base pattern and a transformations set which transform the base pattern to its images. These transformations are part of a transformation group which is the symmetry group of the tiling. To produce a tiling in a systematic way, one has to:

- Choose a symmetry group
- Design a tile compatible with that symmetry group
- Generate a part of the symmetry group (a list of tranformations), apply them to the basic tile to get a list of pictures and group them.

A library for producing tilings can be logically divided into three main modules.

- An algebraic computation module which enables to define and manipulate symmetry groups
- A geometry module which defines the isometries in an effective (numerical) way
- A graphic module: here MLgraph.

If we use ordinary (euclidian) geometry, then we can directly use the MLgraph `transformation` type to represent the isometries. Otherwise, we shall have to represent them as functions of type `point` $\rightarrow$ `point`.
We shall not enter too much into these details here and refer the reader to [4] (Chapter 10) for more information.

[1] The kangaroo shapes we use are taken from drawings by the sculptor Raoul Raba

2.2.2 Tilings in the euclidian plane

Usual tilings of the euclidean plane use transformations that are isometries (combinations of translations, symmetries and rotations). These are linear transformations and therefore easy to perform in MLgraph using values of type transformation.
Given a list of transformations and a pattern which is a value of type picture, the following function produces a tiling by applying each transformation to the pattern and grouping the results.

```
#let make_tiling tr_list pattern =
  group_pictures
   (map (fun tr → transform_picture tr pattern) tr_list);;
make_tiling : transformation list → picture → picture = ⟨fun⟩
```

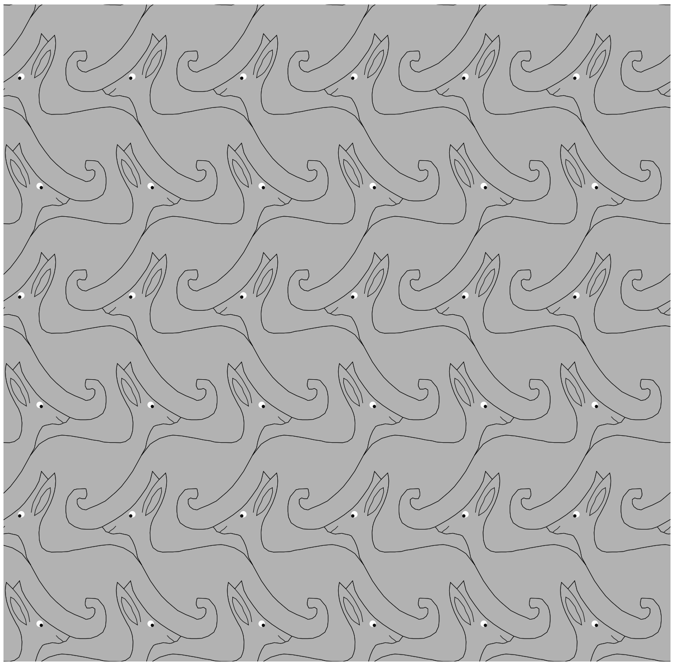


Figure 2.4: A cartesian tiling

For instance, the kangaroo in figure 2.3 can be applied transformations that are compositions of translations and slidings to produce figure 2.4.
Now, if we want a colored tiling[2] we only have to replace the function make_tiling by a function that uses a list of pairs (transformation,color) and a sketch.

```
#let make_colored_tiling trc_list pattern =
  group_pictures
   (map (fun (tr,c) →
```

[2]Unfortunately, here colors are only gray levels

```
                    transform_picture tr
                      (make_draw_picture (lsty,c) pattern))
        trc_list);;
```
make_colored_tiling : (*transformation* $*$ *color*) *list* $\rightarrow$ *sketch* $\rightarrow$ *picture*
= ⟨**fun**⟩

We assume that the variable lsty constains a suitable line style. Using that function, we can obtain figure 2.5.

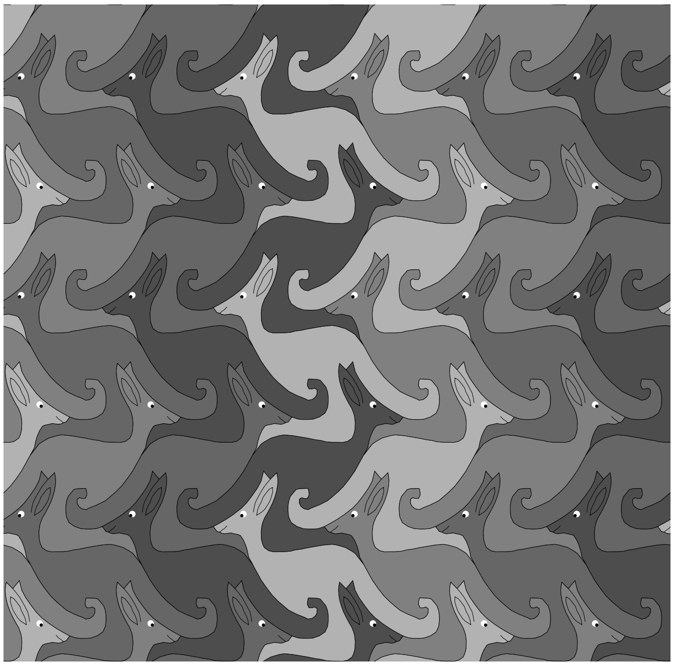


Figure 2.5: A colored tiling

2.2.3 Tiling the hyperbolic plane

If we go from euclidean geometry to other geometries, we have to use different transformations which are not linear transformations that we can simply define for instance as functions of type point $\rightarrow$ point. The basic pattern will be given as a function which, given a transformation and a color, produces a picture. The construction function thus becomes:

```
#let make_general_colored_tiling trc_list pattern =
   group_pictures
     (map (fun (tr,c) -> pattern tr c)
        trc_list);;
```
make_general_colored_tiling
: $(\alpha * \beta)$ *list* $\rightarrow (\alpha \rightarrow \beta \rightarrow$ *picture*$) \rightarrow$ *picture* = ⟨**fun**⟩

In this way we can obtain for instance figure 2.6 which was inspired by Escher's Circle Limit III etching. Of course, the basic kangaroo that we use here is different from the preceding one since the transformations are different (hyperbolic rotations of ordre 3, 3 and 4.

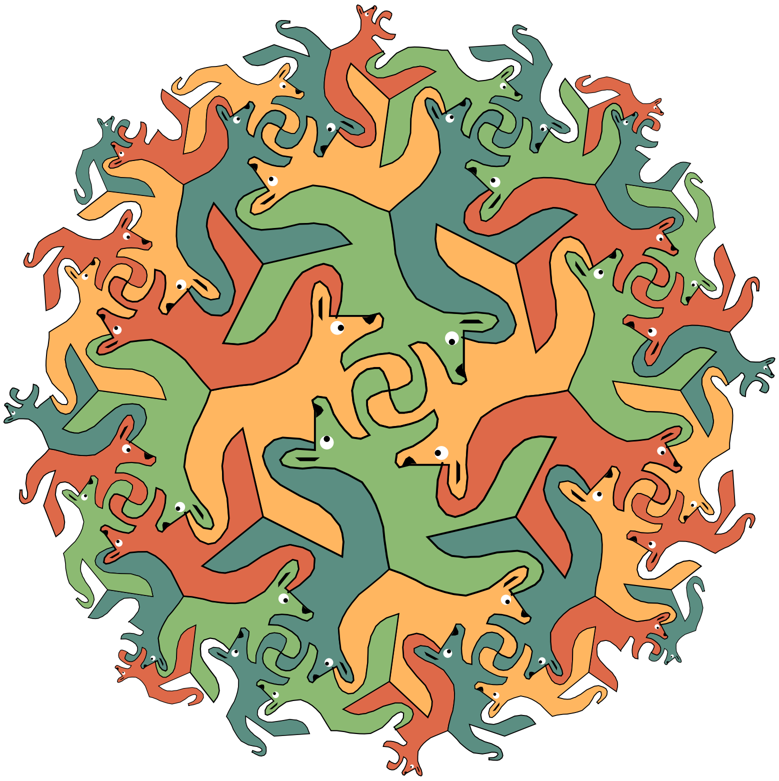


Figure 2.6: A colored hyperbolic tiling

Chapter 3

mlPicTeX a Picture Environment for LaTeX

mlPicTeX is a MLgraph extention for the design of figures which may contain text composed with LaTeX [6]. These figures are described in the MLgraph library language enriched with a set of primitives for drawing lines, arrows, geometrical figures, trees and graphs. This MLgraph extension is now completly integrated in the MLgraph library and can be used directly after loading MLgraph. mlPicTeX allows to describe figures, as MLgraph picture values, in the LaTeX document inside an `mlPic` LaTeX environment in which it is also possible to declare as an MLgraph picture, any piece of LaTeX text, including labels and references. mlPicTeX is compatible with the LaTeX macro definition mechanism and allows to parameterize the mlPicTeX pictures inside macros in order to easily obtain variations on basic pictures.
In the first section we present the mechanism of optional arguments to describe figures. Section 2 shows basic MLgraph extensions for text pictures and surrounding pictures. Section 3 extends the algorithm of automatic placement of tree (see 2.1). Section 4 explains how to build graphs. Section 5 presents the new possibilities offered by mlPicTeX to manipulate LaTeX as MLgraph pictures. Section 6 illustrates the mlPicTeX flexibility and extensibility.

3.1 Optional Arguments

A complete description of simple figures like a circle may be very complex. In addition to its diameter and the coordinates of its center point, it is necessary to have information about the width, shape and color of the circle line, the shape and color of its surface, etc. We have adopted a simple yet powerful and extendable convention for writting mlPicTeX primitives: The generic version of a primitive operation receives a first argument which is a list of optional arguments. The remaining arguments are mandatory.
In order to implement this convention, we defined the type option of named optional values :

```
#type option =
        SOption of string
```

| IOption **of** int
| FOption **of** float
| BOption **of** bool
| POption
| COption **of** color
| LOption **of** string∗float
| DOption **of** int list
| JOption **of** linejoin
| CapOption **of** linecap
| FillOption **of** fillstyle
| FontOption **of** font_style;;

The following functions inject different types into named options (a named option is a value of type string ∗ option) :

string : $\alpha \rightarrow$ string $\rightarrow \alpha \,*$ option
bool : $\alpha \rightarrow$ bool $\rightarrow \alpha \,*$ option
int : $\alpha \rightarrow$ int $\rightarrow \alpha \,*$ option
float : $\alpha \rightarrow$ float $\rightarrow \alpha \,*$ option
option : $\alpha \rightarrow \alpha \,*$ option
color : $\alpha \rightarrow$ color $\rightarrow \alpha \,*$ option
dashPattern : int list $\rightarrow$ string ∗ option
join : linejoin $\rightarrow$ string ∗ option
cap : linecap $\rightarrow$ string ∗ option
fillStyle : fillstyle $\rightarrow$ string ∗ option
lineLabel : string $\rightarrow$ float $\rightarrow$ string ∗ option
font : font_style $\rightarrow$ string ∗ option

For instance color "`background`" yellow builds a named option.
The selector functions extract named values from lists of options, if the name is not found then the default argument is given :

theString : $(\alpha\,*$ option$)$ list $\rightarrow \alpha \rightarrow$ string $\rightarrow$ string
theInt : $(\alpha\,*$ option$)$ list $\rightarrow \alpha \rightarrow$ int $\rightarrow$ int
theFloat : $(\alpha\,*$ option$)$ list $\rightarrow \alpha \rightarrow$ float $\rightarrow$ float
theBool : $(\alpha\,*$ option$)$ list $\rightarrow \alpha \rightarrow$ bool $\rightarrow$ bool
theOption : $(\alpha * \beta)$ list $\rightarrow \alpha \rightarrow$ bool
theColor : $(\alpha\,*$ option$)$ list $\rightarrow \alpha \rightarrow$ color $\rightarrow$ color
theDashPattern : (string ∗ option) list $\rightarrow$ int list $\rightarrow$ int list
theLineJoin : (string ∗ option) list $\rightarrow$ linejoin $\rightarrow$ linejoin
theLineCap : (string ∗ option) list $\rightarrow$ linecap $\rightarrow$ linecap
theFillStyle : (string ∗ option) list $\rightarrow$ fillstyle $\rightarrow$ fillstyle
theLineLabel : (string ∗ option) list $\rightarrow$ option
theFont : (string ∗ option) list $\rightarrow$ font_style $\rightarrow$ font_style

For instance, the mlPicTEX primitive circleGen that draws a circle around a picture receives one mandatory argument and accepts more than 10 optional arguments which includes colors and coefficients for the diameter and the width of the circle line. A simpler primitive called circle describes the same action with all of its arguments taken by default (In fact circle is defined as circleGen [ ]). The two calls of Figure 3.1 describe a circle around a picture p .

circle p

circleGen
[color "background" yellow] p

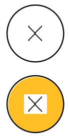


Figure 3.1: Examples of Optional Arguments

The first one describes it with all the default values, while the second one describe this circle with a given color for the background inside the circle. A general drawing uses the following optional arguments :

[
color "background" white;
color "foreground" black;
float "lineWidthCoef" 1.0;
cap Roundcap;
join Roundjoin;
bool "dashedLine" false
] : (string $*$ option) list

The color options select the colors of backgound and foreground. The three following options, "lineWidthCoef", Roundcap, line Roundjoin, describe the line style. dashPattern [4;2] is the default value for the dashed lines if the option "dashedLine" is true.
The others predefined options indicates some picture properties as follow :
options are : bool "closed" false : if true closes the connected parts of the picture skectch;
bool "fill" false : if true closes the picture and fills its "interior";
fillStyle Nzfill : describes the fillstyle, depends of the previous option;
bool "shadowSketch" false : if true draws a shadow shape;
and color "shadow" white : gives the shadow color, depends of the previous option.

3.2 Basic Extensions

We present two basic extensions: the first one to simplify the string to picture conversion and the second one to surround pictures.

3.2.1 Texts

textGen is the general function using the optional argument discipline to convert strings into MLgraph pictures.

text : string $\rightarrow$ picture
textGen : (string $*$ option) list $\rightarrow$ string $\rightarrow$ picture

By default, text uses the following default options :

[font Courier; float "fontSize" 12.0; color "foregroud" black] :
(string ∗ option) list

which define font, font size and foreground color.

3.2.2 Surrounding pictures

Given a picture p, circle p, rectangle p and oval p describe the figure obtained by surrounding p with a circle, a rectangle and an oval respectively (Figure 3.2).

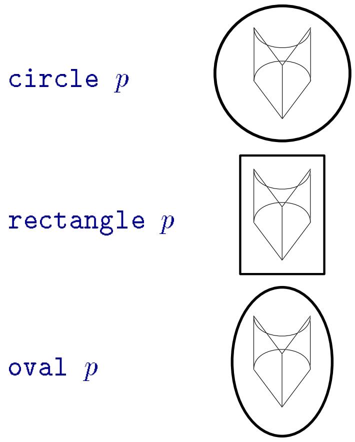


Figure 3.2: Surrounding pictures

The surrounding primitives are used in Figures 3.4 and 3.8, etc. These descriptions accept all line drawing options which and the frameDistanceCoef option which affects the distance between the frame of the picture and the surrounding line and has a default value of 1.

3.3 Placement of Trees

We present an extension of the algorithm of automatic placement (see 2.1).

3.3.1 Types and functions

mlPicTeX defines two primitives for the automatic placement of trees: tree and proofTree. The algorithm to place binary trees in an aesthetic way, described in 2.1, has been extended for general trees.
A tree is a data structure that can contain labels on nodes, leaves and lines. The `tree` primitive of mlPicTeX takes a description of the tree and draws it in a size proportional to the size of its labels:

#type (α,β) tree = Node **of** (α,β) treeRecord | Nil
 and (α,β) treeRecord = {info:α;sons:(α,β) tree list;label:β label}
 and β label = Nolabel | Label **of** β;;

The two main functions to build trees are the following :

tree : (picture, picture) tree $\rightarrow$ picture
treeGen : (string $*$ option) list $\rightarrow$ (picture, picture) tree $\rightarrow$ picture

which draw a description of a tree where nodes and labels are pictures. This new picture has a good space occupation.
A variant of the tree placement algorithm can be used for proof trees. The proofTree function was called in order to produce the picture of Fig 3.3. Its generic form is called proofTreeGen.

```
let leaf info = spacedLeaf info "" in
align_vertically Align_Center[
proofTree
 (node "fx" "lApp"
          [node "f1" "lVar" [leaf "a1"];
           node "x" "lVar" [leaf "b"]]);
proofTree
 (node "lfxffx" "lAbs"
   [node "lxffx" "lAbs"
     [node "ffx" "lApp"
       [node "f2" "lVar" [leaf "a2"];
        leaf "fx"]]])]
```

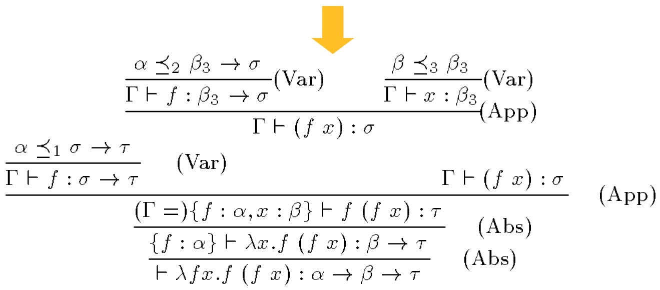


Figure 3.3: An exotic typing proof

Each line between the nodes of a proof tree may have a label. For instance, the strings “`lxffx`” and “`lApp`” correspond respectively to the LaTeX texts $\{f : \alpha\} \vdash \lambda x.f\ (f\ x) : \beta \rightarrow \tau$ and (App).

3.3.2 Optional arguments

The height and width of trees are defined as a value that is proportional to the average size of the subpictures used as nodes. Using the two optional arguments treeHeightCoef and treeWidthCoef, the user can change this proportion and adapts trees to its particular problem. Other possible options for trees include line drawing options and the two following arguments treeLabelPos and labelDistanceCoef for the placement of labels at lines.

3.3.3 An lexical tree example

The following example, Figure 3.4, describes the same tree with two different representations. This kind of tree, called *lexical tree*, is used to recognize words, for example inside a dictionary. We use it with the words “this”, “tea”, “it” and “its”.

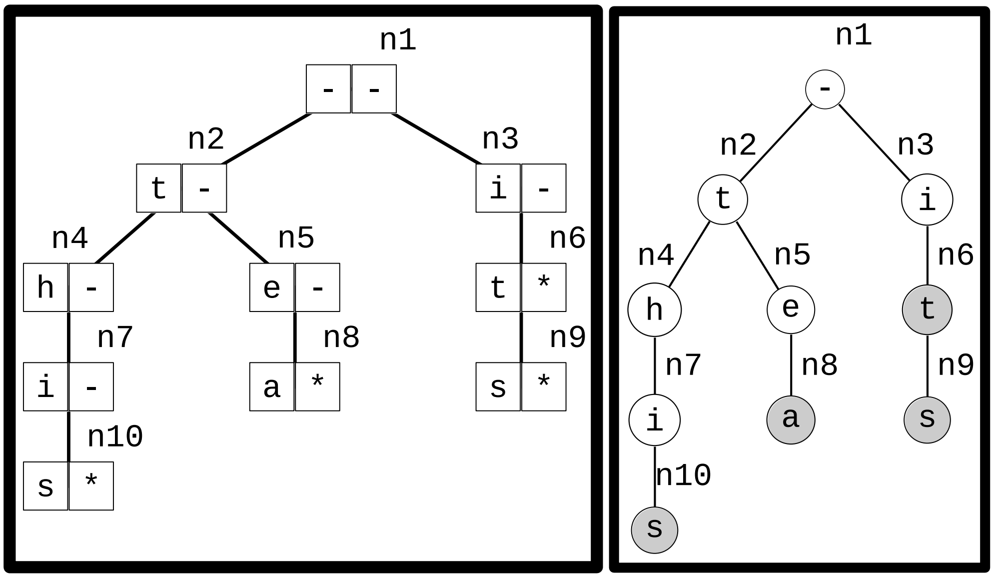


Figure 3.4: Two representations for a lexical tree

Only the decoration of the information changes between the two versions of lexical trees. In the left version, nodes representing the end of a complete word are marked with a star. In the second representation of lexical trees, the drawing node function uses background color to distinguish a possible end of a word. As a consequence of this superposition of information inside nodes, the tree becomes more compact.
These trees are building with the function `n` to describe the nodes.

```
#let l = map (fun x → text x)
          [ "t";"h";"i";"s";"e";"a";"t";"-";"*" ];;

#let l2 = map picture_frame l;;

#let max_frame = merge_frames l2;;

#let reFrame s = force_picture_in_frame max_frame (text s);;

#let makeLabel lab = if lab="" then Nolabel else Label(text lab);;

#let n (s1,s2) lab l =
   Node{info=align_horizontally Align_Center
                 [rectangle (reFrame s1);
```

```
            rectangle (reFrame s2) ]
    ; sons=l;
      label=makeLabel lab};;
```

The function reFrame is used to give the same size to all the information nodes.

```
#let tree_descr_1 =
  n ("-","-") "n1"
   [ n ("t","-") "n2"
      [ n ("h","-") "n4"
         [ n ("i","-") "n7"
            [n ("s","*") "n10" [] ]
         ]
      ;
        * n ("e","-") "n5"
         [ n ("a","*") "n8" [] ]
      ]
   ;
     * n ("i","-") "n3"
      [ n ("t","*") "n6"
         [ n ("s","*") "n9" [] ]
      ]
   ];;
```

The value tree_descr_1 is building following the convention n ("l","-") for a basic node and n ("l","*") for a possible end of word.

```
#let t1 = treeGen [float "treeWidthCoef" 1.5] tree_descr_1;;
```

```
#let grayCircle s = circleGen [color "background" (Gra 0.8)] (text s)
 and normalCircle s = circle (text s);;
```

```
#let n (s1,s2) lab l =
  Node {info=if s2="*"
            then grayCircle s1
            else normalCircle s1;
       sons=l;
       label=makeLabel lab};;
```

```
#let t2 = treeGen [float "treeWidthCoef" 1.5] tree_descr_2;;
```

tree_descr_2 is building with the same declaration than tree_descr_1 with the new definition of the funtion n.

```
#let p =
  let f1 = picture_frame t1 and f2 = picture_frame t2 in
  let mh = merge_frames [f1; f2] in
  (align_horizontally Align_Center
    [rectangleGen[ ]
      (force_picture_in_frame
```

```
        {xmin=f1.xmin;xmax=f1.xmax;ymin=mh.ymin;ymax=mh.ymax} t1);
   rectangleGen[ ]
     (force_picture_in_frame
        {xmin=f2.xmin;xmax=f2.xmax;ymin=mh.ymin;ymax=mh.ymax} t2)
  ]);;
```

p is the final picture in Figure 3.4.
Observe, in the last example, that trees do not make any distinction between left and right sons; when an unique son exists, the line between nodes is vertical. To draw a binary tree, with missing nodes it is sufficient to create an empty brother to the unique son to orientate the drawing.

3.4 Placement of Graphs

mlPicTEX allows to describe a graph easily, by constraints, in the style of *xypic* [7]. The user needs to give the directions for the lines between nodes. The example in Figure 3.5 comes from [3].

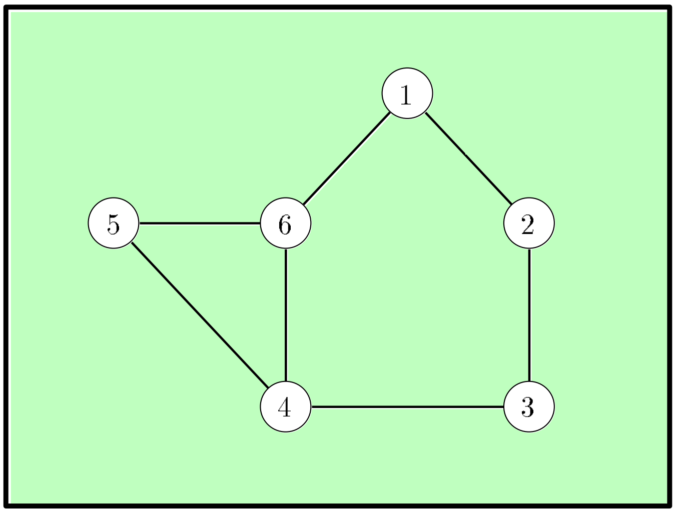


Figure 3.5: Map of the Oahu island (Hawai)

3.4.1 Types and functions

Graphs are represented by values of type **graph**, defined as follows:

```
#type graph =
     Graph of string
   | PGraph of graph*(string*point)list
   | LGraph of graph*((string * option) list * geom_element list list) list
   | TGraph of transformation*graph
   | CGraph of graph*graph;;
```

The following types are used to build graph:

dir for the direction using the cardinal points or directly the rotation angle :

#type dir = Dn | Ds | De | Dw | Dne | Dnw | Dse | Dsw | Deg **of** float;;

lines for lines, curves, symmetric curves or hulls :

```
#type line =
      GLine of (string*option) list*string list
    | GCurve of (string*option) list*(string*dir) list
    | GSCurve of (string*option) list*string*(float*string)list
    | GHull of (string*option) list*string list;;
```

The type graph completely describes the structure of graphs as a set of named edges and a set of lines connecting or surrounding edges. In order to obtain a picture from a description, it is only necessary to add the pictures to be placed at edges and lines. The function `graph` takes a graph and a list associating names and pictures and produces the picture of the graph:

```
graph : graph → (string * picture) list → picture
graphGen : (string * option) list → graph → (string * picture) list
                                           → picture
```

describe graphs with the shape of a regular polygon. polyGraph takes the name of the central point, the name of edges and a list of lines and produce the described graph.

polyGraph : string → string list → line list → graph

The assembleGraphs function joins a set of graphs and a set of independent nodes with positional constraints between nodes :

```
assembleGraphs : graph list → string list → ((string * option) list *
                                              string * dir * string) list
                                           → graph
```

Constraints are pairs of nodes with an orientation. Without any constraint, each node is a graph that is placed at an arbitrary position. A constraint between two isolated graphs concatenates them in order to satisfy the constraint with a straight line. Constraints in both directions between nodes of the same graph are solved with symmetric curves. Constraints affecting only one node are solved as loops.

3.4.2 Optional arguments

The main options for the graphs can played on lines, arrow, labels:
option noLine specifies an invisible line.
string "`arrowDir`" "`F`" is the default value which indicates a forward arrow, backward arrow is "`B`" and a double arrow is "`BF`" or "`FB`".
float arrowLengthCoef 1. gives a coefficent on the arrow length.
float arrowPos 0.5 places the arrow at the middle position.
float loopDir 90. starts a loop of 90. degres. lineLengthCoef 0.5 gives a coefficient for the length of lines.

3.4.3 A finite automata example

The following example, Figure 3.6, describes a typical finite automata.

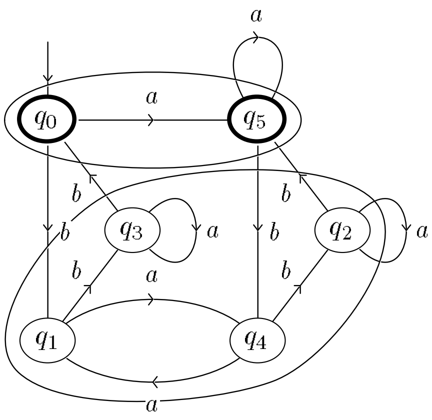


Figure 3.6: Deterministic Finite Automata

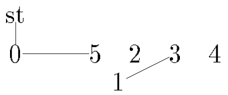
1. We start with a list of unrelated points which will be the nodes of the graph. When no constraint on their position is given, they are just placed anywhere.

```
assembleGraphs []
  ["0";"1";"2";"3";"4";"5";"st"]
  []
```

0 1 2 3 4 5 st

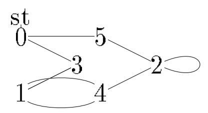
2. Adding a constraint between non connected parts of the graph will positionate them in order to satisfy this constraint with straight lines. Constraints has the form o, a, d, b where o is a list of options, a is a string denoting the source node, d is the direction of the constraint and b is a string denoting target node. The lineLengthCoef modifies distance between edges.

```
assembleGraphs []
  ["0";"1";"2";"3";"4";"5";"st"]
   [[float "lineLengthCoef" 0.5],
          "st",Ds,"0";
    [],"0",De,"5";
    [],"1",Dne,"3"]
```



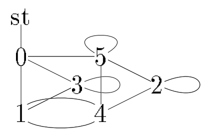
3. Constraints between two different points of the same sub-graph are solved with curves. Constraints between the same point are solved as loops.

```
assembleGraphs []
  ["0";"1";"2";"3";"4";"5";"st"]
   [[float "lineLengthCoef" 0.5],
          "st",Ds,"0";
    [],"0",De,"5";
    [],"1",Dne,"3";
    [],"3",Dnw,"0";
    [],"4",Dne,"2";
    [],"2",Dnw,"5";
    [],"1",Dn,"4";
    [],"4",Ds,"1";
    [],"2",De,"2"]
```



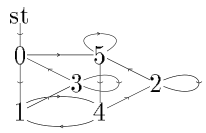
4. When all the points in the graph are connected, the rest of the constraints are solved with curves. These curves are symmetrical with respect to the perpendicular line between the two nodes; their direction is the one specified at the source node and the opposite one at the target node. In some cases these curves may look like straight lines (see for instance, lines $5--4$ and $0--1$).

```
assembleGraphs []
  ["0";"1";"2";"3";"4";"5";"st"]
   [···;
    [],"3",De,"3";
    [],"5",Dn,"5";
    [],"0",Ds,"1";
    [],"5",Ds,"4"]
```



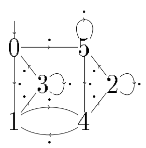
5. The arrows in the graph are specified with the option `string` "`arrowDir`" "`F`". The allowed strings are "`F`" (forward), "`B`" (backward) and "`BF`" (both). This option is added to each oriented line in the graph

```
assembleGraphs []
  ["0";"1";"2";"3";"4";"5";"st"]
   [···;
    [string "arrowDir" "F"],
      "5",Ds,"4"]
```



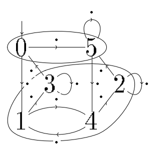
6. The basic graph is completely described. It is possible then to include additional lines or to compose it using others graph description primitives. In order to extend our graph with nodes at label positions in lines, we use the function `addPoints`. In our example, `nodes` is the list of node names to be added as line labels. We also apply a transformation to this graph to avoid the superposition of loops and lines of the previous step.

```
   (addPoints []
    (transformGraph (scaling(1.0,1.3))
      (assembleGraphs [] points lines))
     nodes)
```



7. Another extension to this graph is the introduction of two partitions that will represent states of the same equivalence class. The special line `GHull(`o`,`n`)` draws a continuous line around the sequence n of nodes in a graph. This line can be modified with the list o of line options.

```
addLines
   (addPoints []
    (transformGraph (scaling(1.0,1.3))
      (assembleGraphs [] points lines))
     nodes)
  [GHull([],["0";"5"]);
   GHull([],["1";"3";"2";"4"])]
```



8. Finally, the described graph is drawn with the function `graph` which also add the labels to it producing the picture of Figure 3.6.

`graph` g `labels`

When this file is printed on a PostScript color printer, the background of Figure 3.5 will present a nice green color.

3.5 Integration of Illustrations from LaTeX and MLgraph

This section describes the nature of "ml figures" which are manipulated by mlPicTeX, the way mlPicTeX gets informations from LaTeX and how to obtain final pictures and the document.

3.5.1 Picture description

An ml figure is built from MLgraph primitives and is a value of type picture.The LaTeX boxes are viewed as patches of white ink with exactly the same size as real LaTeX boxes. The Caml program builds this picture and produces a PostScript file where the LaTeX boxes will be rightly overlaid. Figure 3.7 shows this decomposition.

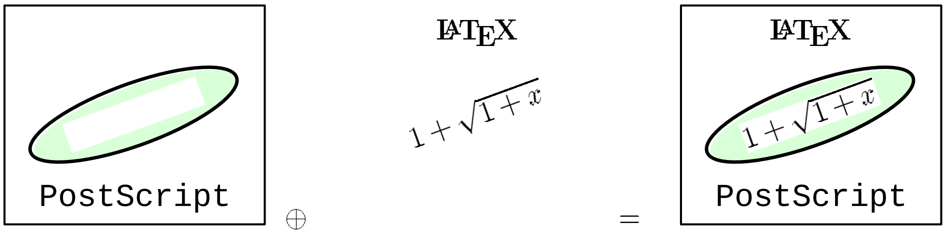


Figure 3.7: Decomposition of picture building by mlPicTeX

One of the main difficulties during the implementation of mlPicTeX was to reconcile the coordinate systems of both PostScript and TeX. Normally, in an ml figure there is no need to give an absolute distance as all distances are given default values by mlPicTeX. However it is always possible to apply an optional distance coefficient to the whole picture or to a particular component. Working in this way, it is rather simple to change the style and font size of a document and recompute ml pictures without modification of their code. Moreover, changes affecting globally the magnification of the document (as in the style *2up.sty*) does not imply any recomputation of ml figures at all.

3.5.2 Getting informations from LaTeX to mlPicTeX

In order to achieve this, we need to manipulate the LaTeX boxes in a special new environment (called mlPic), which captures these exact values and writes them in a new *ml* file during the execution of LaTeX. A simple example is given in Figure 3.8.
Observe, the figure number 3.8 is given by LaTeX in the *ml* file.

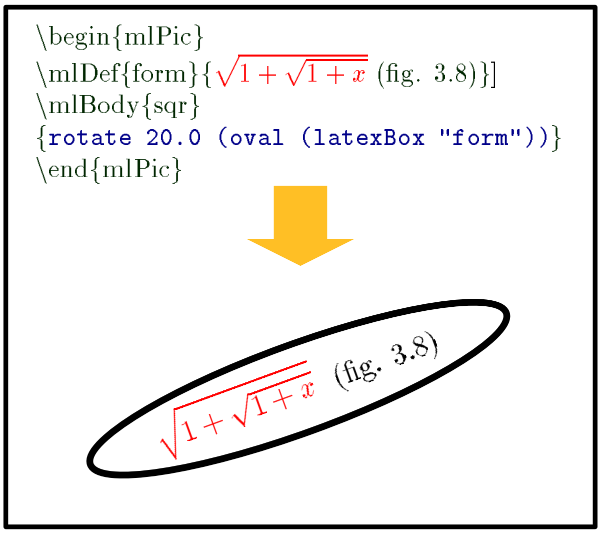


Figure 3.8: An oval frame and its mlPicTeX code

3.5.3 Processing mlPicTeX pictures

The production line is really simple as described in Figure 3.9.
For the simplest case, three steps are necessary:

1. *latex myFile* produces two main files *myFile.pic myFile.ml* and *myFile.dvi*. The mlPic environment creates the *myFile.ml* file from Caml programs encountered inside a \mlBody command. Inside the *dvi* file no picture appears.

2. *camllight ¡ myFile.ml* creates the described pictures. For each picture *p1* there is a file *p1.ps* for the PostScript part of the picture and a file *p1.tex* for its interface to LaTeX.

3. *latex myFile* composes the new document with all ml pictures included. Those Caml programs are evaluated and their execution produce the picture files to be included in the document.

mlpictex is a small Unix script that executes these three steps.

3.5.4 mlPicTeX, the details

The system mlPicTeX is implemented in LaTeX and Caml . In order to use it, you must have these two softwares installed in your system. mlPicTeX can be divided into three units:

- MLgraph.tex. A set of macros written in LaTeX which acts as an output/input communication channel between LaTeX and mlPicTeX. As an output channel, it stores in a Caml file produced by LaTeX the information concerning the boxes to be included in figures and the description of the figures themselves as ml programs. As an input channel, it looks for

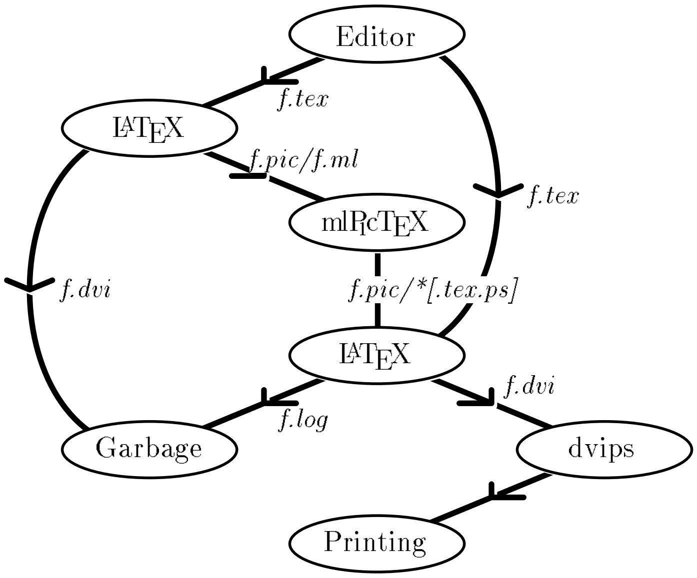


Figure 3.9: Production line of mlPicTeX illustrations

files produced with mlpictex and include them in the document output when they exist.

- mlpictex. A small script that executes Caml toplevel on the ml program produced by LaTeX.
- MLgraph library. Called from the program produced from the document, they compute figures producing PostScript code and tex code to be included in the document.

Although recent versions of LaTeX allows to create and handle a process during the composition of the document, we prefered to implement mlPicTeX as a separate tool just like bibtex. The actions can however be explained as if they were executed synchronously. We will follow now a detailed execution of the example file of Figure 3.10. In line 2, the MLgraph file is included, this make

```
1  \documentstyle...
2  \input{MLgraph}...
3  \begin{document}:...
4  \begin{mlPic}
5   \mlDef{box1}{x^{±3}}
6     ...
7     \mlBody{fig1}{mlexpr}
8  \end{mlPic}
9  \end{document}
```

Figure 3.10: LaTeX file with ml pictures

available to the rest of the document the following commands:

- \begin{mlPic} ... \end{mlPic}, This is the environment in which ml figures are defined (see the LaTeX book [6] for a description of environments). The usage of the following commands outside this environment might have unpredictable results.

- Definition of boxes:
  \mlDef{label}{text},
  \mlDefGen{options}{label}{text}
  and its variants \mlDefBox and \mlVDef. label is the name by which the LaTeX text will be known in Caml . options is a list of optional arguments that modify the layout of text. Optional arguments are discussed in section 3.1. The two other forms are used to define pre-composed horizontal and vertical boxes as ml picture components.

- Description of a picture:
  \mlBody{pictureName}
  {`Caml expression describing the picture`}
  This command is used once in an mlPic environment to describe the picture that the user wants to include in its document and to give the name of the file in which this figure will be stored.

- Directives for the Caml compiler:
  \mlOpen{`"filename"`}
  \mlDir{directive}
  This two commands are used to pass to Caml any directive needed for the evaluation of ml figures in the right environment. These might include the loading of libraries, the declaration of infix operators, etc.

- \mlPut. This is an extension of the \put command of LaTeX which applies a linear transformation to its argument. This command is internally used by the tex code generated with mlPicTeX for the placement of latex boxes into ml pictures. Its variant mlPutg accepts an additional argument which is used to specify colors, to define a clipping zone, etc.

Line 4 of the Figure 3.10 is the start of an ml picture. It prints on the log file of the document a header indicating that output to mlpictex starts there.
In line 5 of Figure 3.10, a new box called `box1` is defined; its contents is the text $x^{\pm 3}$. The actions performed by this command are the creation of a new box using the box creation policy of TeX filled with the LaTeX text $x^{\pm 3}$, and the output to the Caml file of the name and dimensions (but not the contents) of that box. The \mlDefBox variant of this command assumes that its second argument is the number of an existing box that will be used and erased during the creation of the ml picture.
In line 7, the \mlBody command must have as arguments a name for a file to be created in a resource environment and an ml expression of type picture. In this ml expression, the LaTeX box $box1$ can be retrieved as a value of type picture using `latexBox "box1"`. Two files are created : `fig1.ps` to hold the PostScript

part of the picture and `fig1.tex` to hold the LaTeX part of the picture. Finally, in line 8, the `mlPic` environment prints an end of picture trailer in the Caml file and then try to load the figure as produced on a previous run. If this figure is available, it is integrated to the document which can immediately be transformed into postscript and printed with the *dvips* command.

Figure 3.11 shows the corresponding LaTeX file for Figure 3.8. All the figures generated with mlPicTeX are described inside a LaTeX picture environment.

```
\begin{picture}(223.648899999,197.307333042))(0,0)
 \put(0,0){\special{psfile=primer-ml.pic/sqr.ps}}
 \mlPutg{(108.341086666,191.423826375)}{1 0 0 1 }{0}{\box31}
 \mlPutg{(59.2785700231,30.1606974277)}
        {0.939692620786 -0.342020143326 0.342020143326 0.939692620786 }{0}{\box30}
 \mlPutg(5.88350666632,133.118296375)1 0 0 1 0
\end{picture}
```

Figure 3.11: LaTeX file for an mlPicTeX figure

In order to include ml pictures inside other ml pictures you must first create internal ml pictures and store them in boxes; Then, they can be included with the ml primitive \mlDefBox into more complex pictures.

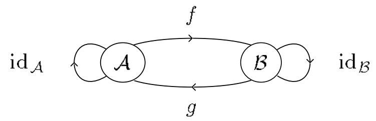
3.6 Flexibility and Extensibility

Firstly, we describe in this section how we can define new LaTeX macros to parameterize mlPicTeX pictures. This feature is convenient to draw small graphs and diagrams. Secondly we give a small example to use new libraries with mlPicTeX. Finally, we show an ml picture (Figure 3.14) built in Caml for the visualization of a tree of values. In that figure, all the \mlDef declarations were automaticly generated by program. This is an interesting approach to the visualization of results of programs.

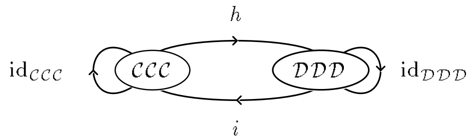
3.6.1 Parameterization of figures

It is possible to parameterize mlPicTeX figures by combining ml pictures and macros in LaTeX. Bellow is the definition of a user-defined macro called \isos used to build small oriented graphs. It has eight parameters. The first is the name of the picture, the next six parameters are the labels of the graph and there is another one for the application of a scaling factor to the graph.

The declaration of `\isos` is the following:



\isos{fig1}{$\mathcal{A}$}{$\mathcal{B}$}{f}{g} {$\mathrm{id}_{\mathcal{A}}$}{$\mathrm{id}_{\mathcal{B}}$}{1.0}



\isos{fig2}{$\mathcal{CCC}$}{$\mathcal{DDD}$}{h}{i} {$\mathrm{id}_{\mathcal{CCC}}$}{$\mathrm{id}_{\mathcal{DDD}}$}{0.7}

Figure 3.12: Two calls of the isos macro

```
\newcommand{\isos}[8]{
\begin{mlPic}
\mlDef{A}{#2}\mlDef{B}{#3}\mlDef{f}{#4}
\mlDef{g}{#5}\mlDef{ida}{#6}\mlDef{idb}{#7}
\mlBody{#1}{
let opt l =
  [string "lineName" l;
   string "arrowDir" "F"] in
let p1 = assembleGraphs [] ["A";"B"]
   [[option "noLine";
     float "lineLengthCoef" 1.5],
               "A",De, "B";
    opt "F", "A",Dnw,"B";
    opt "G", "B",Dse,"A";
    opt "IDA","A",Dw, "A";
    opt "IDB","B",De, "B"]
in
let p2 = addPoints []
  (addPoints
    [float "frameDistanceCoef" 0.5] p1
              ["F",("f",0.5);"G",("g",0.5)])
  ["IDA",("ida",0.5);"IDB",("idb",0.5)]
in
```

```
  graphGen [string "graphStyle" "diagonal"]
    (transformGraph (scaling(#8,#8)) p2)
         ["f",latexBox "f";
          "g",latexBox "g";
          "A",oval(latexBox Ä");
          "B",oval(latexBox "B");
           "ida",latexBox "ida";
          "idb",latexBox "idb"]
}
\end{mlPic}}}
```

There are three steps in this macro. The first step builds in `p1` the skeleton of the graph from the specified constraints. The second step adds labels to the edges of the graph. The third step decorates the skeleton with its labels.

3.6.2 Creating libraries

mlPicTEX allows to include Caml libraries inside the `mlPic` environment. This feature makes it possible to easily extend mlPicTEX with optional and/or user-defined libraries.
Consider the following two definitions inspired by an example of [1] :

```
#let make_h p = set_picture_interfaces p
                 (One_handle ({xc=0.; yc=0.},{xc=10.; yc=0.}),
                  (One_handle ({xc= -1.; yc=0.5},{xc=9.0; yc=0.5})));;

#let create_pict start incr the_end name =
   let rec create_gray start incr the_end r =
     if (le_float start the_end) then r
     else create_gray (start-.incr) incr the_end
           (attach_pictures
              (r, (make_h(latexBoxGen
                          [color "foreground" (Gra (start-.incr))]
                          name))))
   in
   attach_pictures (
                   (create_gray start incr the_end
                      (make_h(make_blank_picture (0.0,0.0)))),
                   (make_h(latexBoxGen
                             [color "foreground" (Rgb (1.0,1.0,1.0))] name)));;
```

The **make**$_h$ function puts two handles on a picture. The function **create_pict** draws n times the same picture in a different level of gray. The attachment of pictures is realized by the function **attach_picture** which identifies the input handle and the output handle of two pictures.
The last sentence to this paper corresponds of the following mlPicTEX description:

```
\begin{mlPic}
\mlOpen{"nzip"}
\mlDef{p1}{This is the End.}
\mlBody{f3}{
create_pict 1.0 0.05 0.  "p1"
}
\end{mlPic}
```

Figure 3.13: LATEX text manipulation

3.6.3 Building mlPicTEX from Caml

We try to build a picture, corresponding to a tree, which represents all the integer values which converge in less than nine steps for the "Syracuse conjecture". The definition is as follows:

$$\begin{array}{lcrl} u_{n+1} & = & 1 & \text{if } u_n = 1 \\ u_{n+1} & = & \frac{u_n}{2} & \text{if } u_n \text{ is even} \\ u_{n+1} & = & 3 * u_n + 1 & \text{if } u_n \text{ is odd} \end{array}$$

The Syracuse conjecture is "For all $u_0 > 0$ this series converge to 1".

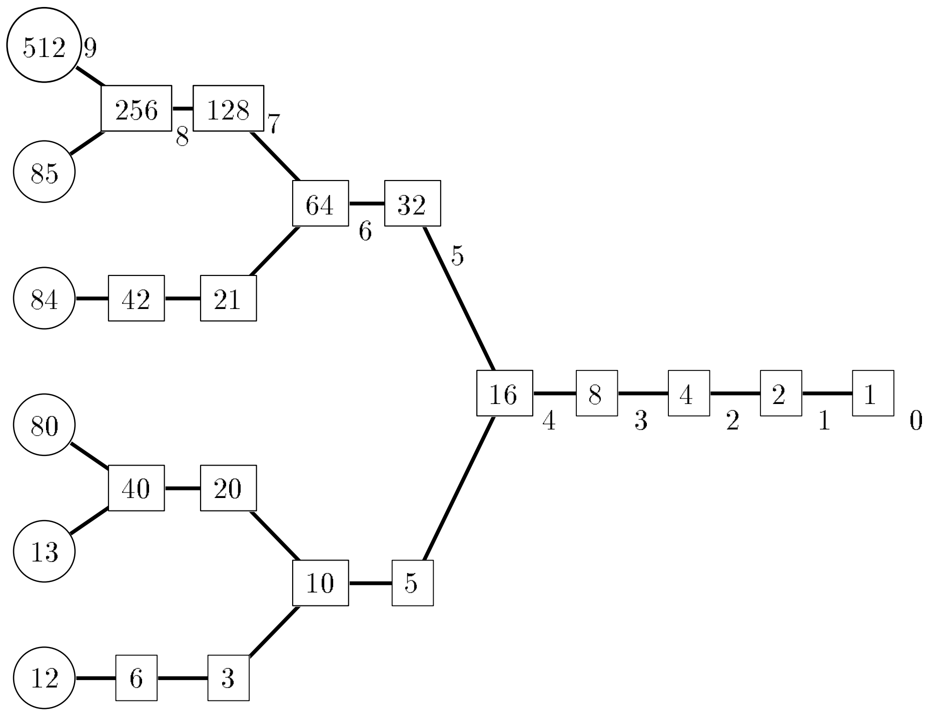


Figure 3.14: Convergence in at most nine steps

The labels indicate the number of steps needed to converge to 1. For example the number 3 converges in seven steps to 1. This number list is computed in

Caml , translated to mlPicTeX, and then processed by mlPicTeX, which calls once again Caml . This is the way to visualize a Caml value in a LaTeX document (the corresponding picture is built with the LaTeX fonts). To obtain this horizontal tree we make a (90) rotation on each node and at the end a (270) rotation on the tree.

Chapter 4

Installing and using MLgraph

4.1 Availability

The complete MLgraph distribution resides on the machine `ftp.ens.fr`. The distribution files can be transfered by anonymous FTP:

Host:	`ftp.ens.fr`
Login name:	`anonymous`
Password:	your email address
Directory:	`pub/unix/lang/MLgraph`
Files:	see the index in file `INDEX`

More information on the MLgraph library is also available on the author's WWW pages: `http://cadillac.lip6.fr/~emmanuel` and `http://www.ens.fr/~cousinea`. The email address `MLgraph@ens.fr` corresponds to an alias to contact the authors.

Actually MLgraph system is available in several versions: for Caml Light (from version 0.6) and Objective Caml (from version 1.0) for the same machines than Caml (i.e. Unix, MacIntohses and PCs). To use mlPicTeX in conjunction with LaTeX it is necessary to install on your machine LaTeX.

4.2 Installation Instructions

See the file INSTALL.

4.3 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.

4.3.1 interactive session

The library is contained in only one file. This choice facilitates its use. The following sessions show how to build a ML picture and the corresponding

PostScript file.
In Caml Light :

```
% camllight
>       Caml Light version 0.6

##open "MLgraph";;
#load_object "MLgraph";;
- : unit = ()
#make_blank_picture;;
- : float * float -> picture = <fun>
#text;;
- : string -> picture = <fun>
#let p = text "salut tout le monde!!!";;
p : picture = <abstract>
#let p1 = translate (250.0,300.0) p;;
p1 : picture = <abstract>
#eps_file p1 "ex";;
- : unit = ()
#
```

and in Objective Caml :

```
% ocaml
        Objective Caml version 1.01

# open Mlgraph;;
# #load "mlgraph.cma";;
# make_blank_picture;;
- : float * float -> Mlgraph.picture = <fun>
# text;;
- : string -> Mlgraph.picture = <fun>
# let p = text "salut tout le monde!!!";;
val p : Mlgraph.picture = <abstr>
# let p1 = translate (250.0,300.0) p;;
val p1 : Mlgraph.picture = <abstr>
# eps_file p1 "ex";;
- : unit = ()
#
```

In these two cases, a PostScript file `ex.ps` will be created. The main difference between Caml Light version and Objective Caml is the representation of types. Objective Caml shows the module name (see the type of function make_blank_picture, where appears the module name Mlgraph.
If you have chosen another directory than the usual Caml directory (`/usr/local/lib/caml-light` or `/usr/local/lib/ocaml/`), change the graphics directory as follow :

```
% camllight
```

```
>       Caml Light version 0.6


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

You can include the caml-light file `mlgraph.ml` (`mlgraph` for Objective Caml) which contains a template.

4.3.2 Command line

If you prefer to work by command line, you need to add

in Caml Light the `MLgraph.zo` file to your own code :

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

where `work.ml` is your working file

and in Objective Caml the file `mlgraph.cma` in the following way :

```
ocaml -o work.exe mlgraph.cma work.ml
```

Be careful, you are not independent of the `MLgraph.lib` directory which contains font descriptions, several commands and the PostScript headers (see 4.5 section to build standalone applications).

4.4 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 description font. The type font_description is defined as follows :

```
#type font_description =
    { font_descr_filename : string;
      mutable font_descr_name : string ;
      mutable font_descr_height : float ;
      mutable font_descr_width : float ;
      mutable font_descr_descr : float vect;
      mutable font_descr_descr_bbox : ((float * float ) * (float * float)) vect
    };;
```

and the main font manipulation functions are :

```
add_font : font_style * font_description → unit
remove_font : font_style * font_description → unit
```

For example, if you want to use the **Ohlfs** font, then :

1) create the font description, called `Ohlfs.fnt`, 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",
              {font_descr_filename="your_file";
               font_descr_name="Ohlfs";
               font_descr_height=12.0;
               font_descr_width=12.0;
               font_descr_descr=[| |];
               font_descr_bbox=[| |]});;
   ```

   to create its entry point.

All font descriptions are automatically loaded at first use, but you can load them from ML programs (see 4.5).
To visualize some font informations, the following functions :

```
print_font_list : unit → unit
print_info_all_fonts : unit → unit
```

display the name of all fonts and their state.

4.5 Building Standalone Applications

It is necessary to link in your application the font description files needed (in a ML format) and the PostScript header file (also in a ML format) to be completely independant of the font descriptions and PostScript header directories. These different files are localized in the `Lib` directory inside the `MLgraph.lib` directory.
The ML font description files are generated from their ASCII description file and using the add_font function.
The function modify_body_prelude takes a string list which contains the PostScript prelude code and uses it as the new prelude.

modify_body_prelude : string list → unit

If the string list is empty, MLgraph loads the PostScript header file when it translates a picture value in PostScript. In the other case, the string list is used as PostScript prelude.
The file `xHeader.ml` contains the ML definition (call to modify_body_prelude) of the default PostScript header.
For example, in Caml Light, you need to compile your file `work.ml` as follow :

```
camlc -o work.exe MLgraph.zo -IMLgraph.lib/Lib xHeader.ml xHelv.ml work.ml
```

to build a standalone application.

4.6 Using MLgraph pictures in LaTeX

See section 3.5. For Unix machines, it is usefull to modify the `TEXINPUTS` to load properly the file `MLgraph.tex` and the `PATH` variables to use the `mlpictex` shell script.

4.7 Testing Pictures

The function latex_mode applied on the () value returns a boolean which indicates if your piece of MLgraph code is extracted from a LaTeX file. In this case you can use the latexBox function. If you wish to test your MLgraph program before including it in your LaTeX document, it is necessary to remplace the calls to latexBox by the text function. For that, add at the beginning of your program the following text :

```
#let latexBox s =
  if latex_mode() then latexBox s
  else text s;;
```

4.8 Using *epsf* Macro

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}
\mbox{\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 : Tutorial and Cookbook.* Addison-Wesley, 1985.

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

[3] AHO, A., AND ULLMAN, J. *Concepts fondamentaux de l'Informatique.* DUNOD, 1993.

[4] COUSINEAU, G., AND MAUNY, M. *Approche Fonctionnelle de la Programmation.* EdiScience, 1995.

[5] GRÜNBAUM, B., AND SHEPHARD, G. *Tilings and patterns.* Wiley and Sons, 1987.

[6] LAMPORT, L. *LaTeX User's Guide and Reference Manual.* Addison-Wesley, 1986.

[7] ROSE, K. H., AND MOORE, R. R. *Xy-pic Reference Manual*, 3.0 ed. DIKU, University of Copenhagen, Universitetsparken 1, DK–2100 København Ø, 1995.

Index of concepts

Index of functions and types

Index of mlPicTeX types and functions

Index of mlPicTEX option

List of Figures