functional METAPOST

User's manual

Joachim Korittky

April 11, 2002

Abstract

functional METAPOST was created by Joachim Korittky as his diploma thesis at the Universität Bonn in 1998.
This text is a translation of the chapters 3 and 4 and the appendices of this thesis and gives a description of the *functional* METAPOST language.

Contents

List of Figures

Chapter 1

Introduction to *functional* METAPOST

functional METAPOST is embedded into the programming language Haskell. This means that the full power of Haskell is available for calculations and data manipulations. *Functional* METAPOST provides the additional types and functions which allow to generate and manipulate graphics objects:

- The basic data type for expressions representing pictures is *Picture*.
- The language has functions to generate atomic pictures (e.g. text) and functions to combine pictures.
- There are also functions to generate frames around pictures. Different frame styles are provided.
- A picture can be amended by paths.
- The language has the possibility to draw areas, apply affine transforms to pictures and use equations to define implicitly relative positions and geometric relations.
- Properties of paths and pictures (e.g. colors or pencils) are represented as attributes. Attributes have sensible defaults and can be changed by attribute functions.

The bounding box is in first approximation a minimal enclosing rectangle rectangle around the picture. The expression

$$text\ \texttt{"red"}\ \#\ setColor\ red$$

is a first example for the application of an attribute function. It generates red text. For notational convenience the operator (#) is defined as "backward" application

$$\begin{array}{lcl} (\#) & :: & a \to (a \to b) \to b \\ a\ \#\ f & = & f\ a \end{array}$$

The names of attribute functions start with *set* (change attribute) or *get* (read attribute).

This chapter is introductory in order to allow a fast start with *functional* METAPOST, it does not introduce all functions. A complete list is given in chapter 3.

Also, we will start describing *functional* METAPOST as it should be, without consideration of some limitations imposed by Haskell. Section 1.16 discusses these limitations and how to bypass them.

1.1 Atomic pictures

Atomic pictures are the basic building blocks of more complex pictures. Two such atomic constructs are a) the inclusion of arbitrary LaTeX expressions:

$$\mathit{tex} \quad :: \quad \mathit{String} \rightarrow \mathit{Picture}$$

and b) the generation of an empty rectangle of given width and height:

$$\mathit{space} \quad :: \quad \mathit{Numeric} \rightarrow \mathit{Numeric} \rightarrow \mathit{Picture}$$

It is easy to define other useful functions, e. g. for the LaTeX math mode

$$\begin{aligned} &\mathit{math} && :: \quad \mathit{String} \rightarrow \mathit{Picture} \\ &\mathit{math}\ p && = \quad \mathit{tex}\ (\texttt{"\$"} +\!\!+\ p +\!\!+ \texttt{"\$"}) \end{aligned}$$

or commands to create empty spaces (with names similar to their LaTeX equivalents)

$$\begin{aligned} &\mathit{hspace}, \mathit{vspace} && :: \quad \mathit{Numeric} \rightarrow \mathit{Picture} \\ &\mathit{hspace}\ n && = \quad \mathit{space}\ n\ 0 \\ &\mathit{vspace}\ n && = \quad \mathit{space}\ 0\ n \end{aligned}$$

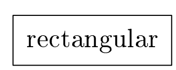
This one will be usefull too:

$$\begin{aligned} &\mathit{empty} && :: \quad \mathit{Picture} \\ &\mathit{empty} && = \quad \mathit{space}\ 0\ 0 \end{aligned}$$

1.2 Frames

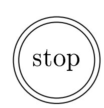
Frames adopt their size automatically to the framed picture. The expression

$$\mathit{box}\ (\mathit{tex}\ \texttt{"rectangular"}) \quad :: \quad \mathit{Picture}$$



generates a rectangular frame around the text "rectangular"[1] A multiple application generates multiple frames:

$$\mathit{circle}\ (\mathit{circle}\ (\mathit{tex}\ \texttt{"stop"})) \quad :: \quad \mathit{Picture}$$



Other frame styles exist, among them *oval*, *triangle* and *rbox*. The distance between frame and picture can be changed by:

$$\mathit{setDX} \quad :: \quad \mathit{Double} \rightarrow \mathit{Picture} \rightarrow \mathit{Picture}$$

and

$$\mathit{setDY} \quad :: \quad \mathit{Double} \rightarrow \mathit{Picture} \rightarrow \mathit{Picture}$$

[1] We will often show a picture together with the expression describing the picture, separated by a vertical or horizontal line.

The expression

$$\begin{array}{ll} \mathit{rbox}\ 5\ (\mathit{tex}\ \texttt{"rounded\textvisiblespace box"}) & \\ \#\ \mathit{setDX}\ 10 & \\ \#\ \mathit{setDY}\ 5 & :: \ \mathit{Picture} \end{array}$$

generates a frame with rounded corners of radius 5 and with a slightly enlarged distance to the text. Using a circle frame we can define a dot, i.e. a small filled circle:

$$\begin{array}{lll} \mathit{dot} & :: & \mathit{Picture} \\ \mathit{dot} & = & \mathit{circle}\ \mathit{empty} \\ & & \quad \#\ \mathit{setBGColor}\ \mathit{black} \\ & & \quad \#\ \mathit{setDX}\ 0.75 \end{array}$$

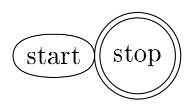
1.3 Combination of pictures

The most common combinators set two pictures side by side or on top of the other:[2]

$$\begin{array}{lll} (\square\!\square) & :: & \mathit{Picture} \to \mathit{Picture} \to \mathit{Picture} \\ (\genfrac{}{}{0pt}{}{\square}{\square}) & :: & \mathit{Picture} \to \mathit{Picture} \to \mathit{Picture} \end{array}$$

Consider the expression

$$\mathit{oval}\ \texttt{"start"}\ \square\!\square\ (\mathit{circle}\ (\mathit{circle}\ \texttt{"stop"}))$$



A new picture is created in which the two pictures are horizontally arranged without additional space. The combinators $(\square\!\square)$ and $(\genfrac{}{}{0pt}{}{\square}{\square})$ are associative:

$$\begin{array}{lll} a \ \square\!\square\ (b\ \square\!\square\ c) & \equiv & (a\ \square\!\square\ b)\ \square\!\square\ c \\ a\ \genfrac{}{}{0pt}{}{\square}{\square}\ (b\ \genfrac{}{}{0pt}{}{\square}{\square}\ c) & \equiv & (a\ \genfrac{}{}{0pt}{}{\square}{\square}\ b)\ \genfrac{}{}{0pt}{}{\square}{\square}\ c \end{array}$$

Mathematically speaking, the empty picture is a neutral element:

$$\begin{array}{lll} a\ \square\!\square\ \mathit{empty} & \equiv & a \\ a\ \genfrac{}{}{0pt}{}{\square}{\square}\ \mathit{empty} & \equiv & a \end{array}$$

and we have two semigroups $((\square\!\square), \mathit{empty})$ and $((\genfrac{}{}{0pt}{}{\square}{\square}), \mathit{empty})$.

Sometimes we want a small distance between the pictures:

$$\begin{array}{lll} (\square\,\square) & :: & \mathit{Picture} \to \mathit{Picture} \to \mathit{Picture} \\ a\ \square\,\square\ b & = & a\ \square\!\square\ \mathit{hspace}\ 8\ \square\!\square\ b \end{array}$$

$$\begin{array}{lll} (\genfrac{}{}{0pt}{}{\square}{\raise1pt\square}) & :: & \mathit{Picture} \to \mathit{Picture} \to \mathit{Picture} \\ a\ \genfrac{}{}{0pt}{}{\square}{\raise1pt\square}\ b & = & a\ \genfrac{}{}{0pt}{}{\square}{\square}\ \mathit{vspace}\ 8\ \genfrac{}{}{0pt}{}{\square}{\square}\ b \end{array}$$

[2]For better readability, operators are shown here in a graphical form. Appendix A shows the ASCII representation to be used in real source code.

Fig. 1.1 shows a picture generated by text, frames and combinators only. The function *rbox20* is defined as

$$rbox20\ a \quad = \quad rbox\ 20\ a\ \#\ setDX\ 8\ \#\ setDY\ 6$$

and generates a frame with round corners of radius $\leqslant 20$.

rbox20 (*tex* `"set␣of␣all␣languages"`
$\genfrac{}{}{0pt}{}{\square}{\square}$ *rbox20* (*tex* `"rec.␣enum.␣languages␣or␣type␣0"`
$\genfrac{}{}{0pt}{}{\square}{\square}$ *rbox20* (*tex* `"decidable␣languages"`
$\genfrac{}{}{0pt}{}{\square}{\square}$ *rbox20* (*tex* `"context␣sensitive␣or␣type␣1"`
$\genfrac{}{}{0pt}{}{\square}{\square}$ *rbox20* (*tex* `"context␣free␣or␣type␣2"`
$\genfrac{}{}{0pt}{}{\square}{\square}$ *rbox20* (*tex* `"regular␣or␣type␣3"`))))))

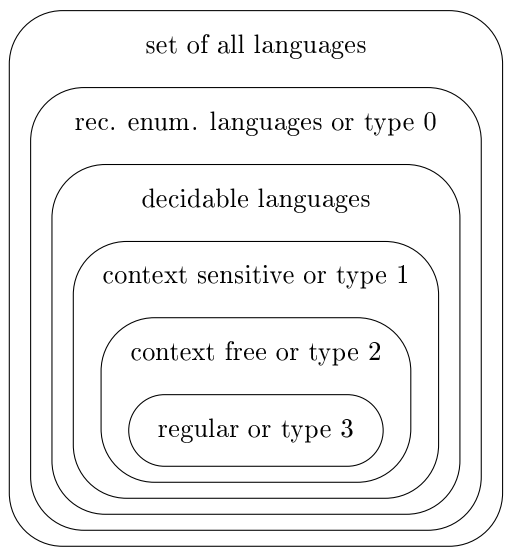


Figure 1.1: A picture with frames and combinators, from [Sch97].

METAPOST uses as basic unit PostScript points which correspond to 1/72 inch. Some predefined constants allow the use of other units

$$\begin{array}{lcl} mm, pt, cm & :: & Numeric \\ mm & = & 2.83464 \\ pt & = & 0.99626 \\ cm & = & 28.34645 \end{array}$$

This allows us to write $2 * cm$ to define a length of 2 cm.

The combinators have generalizations for more than two pictures:

$$
\begin{array}{lll}
\mathit{row} & :: & [\mathit{Picture}] \rightarrow \mathit{Picture} \\
\mathit{column} & :: & [\mathit{Picture}] \rightarrow \mathit{Picture}
\end{array}
$$

Both functions could be defined as follows:[3]

$$
\begin{array}{lll}
\mathit{row} & = & \mathit{foldr}\ (\square\!\square)\ \mathit{empty} \\
\mathit{column} & = & \mathit{foldr}\ (\genfrac{}{}{0pt}{}{\square}{\square})\ \mathit{empty}
\end{array}
$$

Also useful are variants of these functions which allow space between the pictures:

$$
\begin{array}{lll}
\mathit{rowSepBy} & :: & \mathit{Numeric} \rightarrow [\mathit{Picture}] \rightarrow \mathit{Picture} \\
\mathit{columnSepBy} & :: & \mathit{Numeric} \rightarrow [\mathit{Picture}] \rightarrow \mathit{Picture}
\end{array}
$$

defined as:

$$
\mathit{rowSepBy}\ n \quad = \quad \mathit{foldr}\ (\lambda a\ b \rightarrow a\ \square\!\square\ \mathit{hspace}\ n\ \square\!\square\ b)\ \mathit{empty}
$$

Fig. 1.2 shows our traffic light example. We define a function *light*, which generates text with a given background color, framed in distance 10 by a circle. The three lights are combined in a column with distance 10 and finally framed by a box with distance 10.

```
let light s c = circle (tex s)
                # setDX 10
                # setBGColor c in
box (columnSepBy 10
      [light "R" red,
       light "O" yellow,
       light "G" green])
 # setBGColor black
 # setDX 10
 # setDY 10
```

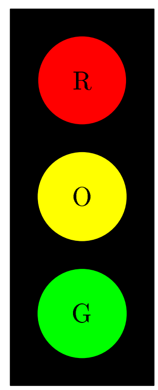


Figure 1.2: A traffic light.

Sometimes we need a two-dimensional arrangement of pictures

$$
\mathit{matrix} \quad :: \quad [[\mathit{Picture}]] \rightarrow \mathit{Picture}
$$

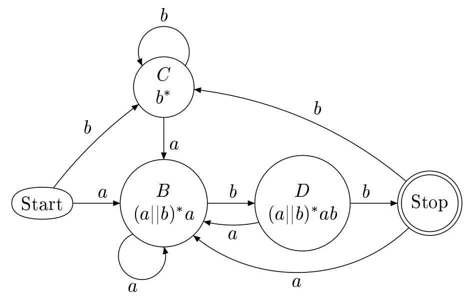
The function *matrix* generates columns and rows. Every column has the width of the picture with the largest width in it. Analogously, every row has the height of the highest picture in it. Then the pictures are centered in the so-defined rectangular cells. There is also a variant with additional space:

[3]The function *foldr* extends a binary operator to a list:

$$
\begin{array}{lll}
\mathit{foldr}\ f\ z\ [\,] & = & z \\
\mathit{foldr}\ f\ z\ (x : xs) & = & f\ x\ (\mathit{foldr}\ f\ z\ xs)
\end{array}
$$

```haskell
matrixSepBy :: Numeric -> Numeric -> [[Picture]] -> Picture
```

Let us now use all this to give the first part of a graph of a finite state machine (from[Hob92]). The pictures for the initial and final state came already as examples. The function *stk* generates two lines of LaTeX, one on top of the other.




```haskell
stk     :: String -> String -> Picture
stk a b = math ("\\matrix{" ++ a ++ "\\cr " ++ b ++ "\\cr}")
```

The pictures for the states are generated by frames and combined with the function *matrixSepBy*. The arrows between the states are still missing.

```haskell
automat = matrixSepBy 30 20
            [[empty, circle (stk "C" "b^*")          # setName "C"],
             [oval (tex "Start")                     # setName "start",
              circle (stk "B" "(a||b)^*a")           # setName "B",
              circle (stk "D" "(a||b)^*ab")          # setName "D",
              circle (circle (tex "Stop"))           # setName "stop"]]
```

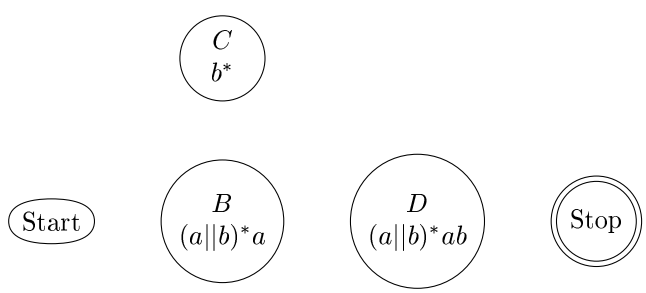
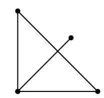


Figure 1.3: First part of a finite state machine: the states are aligned by the function *matrixSepBy*.

1.4 Paths

Paths are besides pictures the most important objects in *functional* METAPOST. Different from pictures, they do not have a bounding box but are drawn on pictures. A path consists of points and connections between them. The function *vec* defines a point by its coordinates and the path constructor (--) connects the points. The points are emphasized in this example:

$vec\ (20,20)$ -- $vec\ (0,0)$ -- $vec\ (0,30)$
--$vec\ (30,0)$ -- $vec\ (0,0)$



The path constructor (--) generates a path segment. There are four different kinds of path constructors:[4]

(--)	A straight segment.
(..)	A curved segment.
(---)	A straight segment with endings as smooth as possible.
(...)	A curved segment with as few as possible turning points.

The course of a path is determined not only by its points but essentially by the kind of connections between them. The three paths in Fig 1.4 are different but connect the same points.

The path segments between the points z_0 and z_1 or z_1 and z_2 are all created by the path constructor (..), but are different, too.[5] A closed, cyclic path can be obtained by using the expression *cycle*[6] as final point.

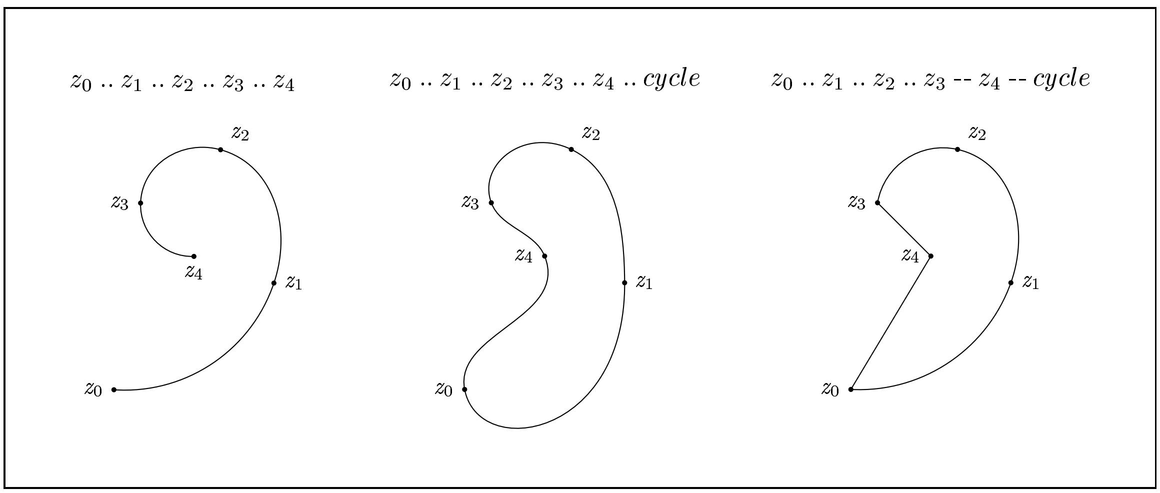


Figure 1.4: The effect of different path connections on the same set of points.

Attribute functions allow further manipulations of paths. Every segment has its own set of attributes. For the curved connection types (..) and (...) we can use the functions

[4]Curved segments are constructed by BÉZIER splines. Consider points $z_0, z_1, ...z_n$. Then there exist auxiliary points z_k^+ and z_{k+1}^- such that the cubic spline between z_k and z_{k+1} is given by the BERNSTHEĬN polynomial

$$z(t) = B(z_k, z_k^+, z_{k+1}^-, z_{k+1}; t) = (1-t)^3 z_k + 3(1-t)^2 t z_k^+ + 3(1-t)t^2 z_{k+1}^- + t^3 z_{k+1}$$

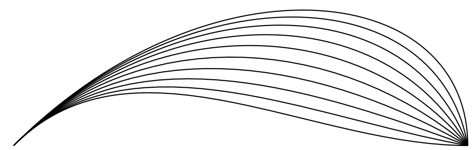
for $0 \leqslant t \leqslant 1$. Compare [Fel92]. Usually the auxiliary points are chosen such that the path segments are C^1 continuous at the transition points

[5]This is a result of the properties (C^1 continuity) of BÉZIER splines.

[6]See Appendix A for the *cycle* keyword!

setStartAngle and *setEndAngle* to set the angle with which a path segment starts or ends at a point. This is illustrated in Fig. 1.5. The path constructors have a higher precedence as the (#) operator. Therefore, $a\,..\,b\,..\,c \# setStartAngle\ d$ is equivalent to $(a\,..\,b\,..\,c) \# setStartAngle\ d$.

$$
\begin{array}{l}
[\mathit{vec}\ (0,0)\ ..\ \mathit{vec}\ (6 * \mathit{cm}, 0) \\
\quad \#\ \mathit{setStartAngle}\ 45 \\
\quad \#\ \mathit{setEndAngle}\ (-10 * a) \\
\mid a \leftarrow [0\,..\,9]]
\end{array}
$$



$$
\begin{array}{l}
[\mathit{vec}\ (0,0)\ ..\ \mathit{vec}\ (6 * \mathit{cm}, 0) \\
\quad \#\ \mathit{setStartAngle}\ 45 \\
\quad \#\ \mathit{setEndAngle}\ (10 * a) \\
\mid a \leftarrow [0\,..\,7]]
\end{array}
$$

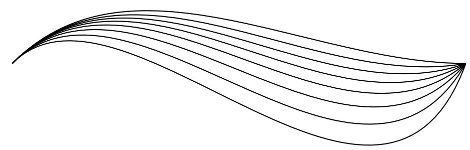


Figure 1.5: Start and end angle of path segments can be set.

Another possibility to change the appearance of the Bézier splines are the functions *setStartVector* and *setEndVector*. This is demonstrated in Fig. 1.6, which also shows the path connector (...) in action. Different from (..), this connector avoids turning points.

$$
\begin{array}{l}
(z_0\ ..\ z_1\ \#\ \mathit{setStartVector}\ (\mathit{vec}\ (0,1)) \\
\qquad\quad \#\ \mathit{setEndVector}\ (\mathit{vec}\ (1,0))) \\
..z_2\ \#\ \mathit{setEndVector}\ (\mathit{vec}\ (0,-1)) \\
\\
(z_0\ ...\ z_1\ \#\ \mathit{setStartVector}\ (\mathit{vec}\ (0,1)) \\
\qquad\quad \#\ \mathit{setEndVector}\ (\mathit{vec}\ (1,0))) \\
...z_2\ \#\ \mathit{setEndVector}\ (\mathit{vec}\ (0,-1))
\end{array}
$$

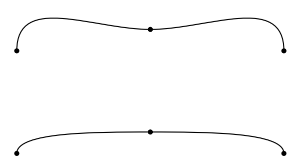


Figure 1.6: The difference between (..) and (...).

Every path segment can have one or more labels. Labels can be arbitray pictures. This is done by the function

$$\mathit{setLabel} \quad :: \quad \mathit{Numeric} \rightarrow \mathit{Dir} \rightarrow \mathit{Picture} \rightarrow \mathit{Path} \rightarrow \mathit{Path}$$

The first parameter defines the position along the path. Its value must be from the interval $[0;1]$, where 0 denotes the begin and 1 the end of the path. The second parameter describes the alignment of the label relative to the path and the third parameter is the picture of the label. Fig 1.7 shows an example.

```
(vec (0,40) .. vec (10,50)
  .. vec (15,40) .. vec (10,7)
  .. vec (15,0) .. vec (25,7))
 # setLabel 0 N (tex "start")
 # setLabel 0 C dot
 # setLabel 0.5 W (tex "middle")
 # setLabel 0.5 C dot
 # setLabel 1 W (tex "end")
 # setLabel 1 C dot
```

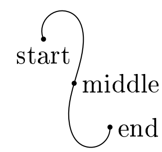


Figure 1.7: Paths can include arbitrary labels.

In *functional* METAPOST, paths are drawn on top of pictures, or pictures are "decorated" by paths. A list of paths is added to a picture by

$$draw \quad :: \quad [Path] \to Picture \to Picture$$

We have seen some pictures of bare paths. How are they created? One possibility is to draw them on an empty picture. This way, the final picture gets the bounding box of the *empty* picture since the drawing of paths does not change the bounding box of a picture. This can be corrected with the function

$$setTrueBoundingBox \quad :: \quad Picture \to Picture$$

which creates a rectangular minimal enclosing bounding box. The following expression creates a picture of the paths *ps*:

$$setTrueBoundingBox\ (draw\ ps\ empty)$$

We will see a shorter notation for this expression in section 1.16.

1.5 Names

Up to now we always defined points using the *vec* function. There is another way to reference to points of a picture.

Every picture has nine predefined reference points which mark its bounding box. These points have names N, NE, E, SE, S, SW, W, NW, according to the compass points and C for the center of the picture. Fig. 1.8 shows reference points and bounding box of a picture. Reference points can be used in path descriptions etc. by using

```
  tex "let␣this␣be"
□ tex "a␣picture"
□
```

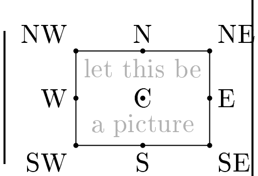


Figure 1.8: Every picture has nine reference points.

$$ref \quad :: \quad Name \rightarrow Point$$

The following expression uses the reference points of the picture *tex* "!" to draw a path

$$\begin{array}{l} draw\ [ref\ \ NE + vec\ (2,0)\ \text{--}\ ref\ \ S - vec\ (0,2) \\ \qquad\quad \text{--}\ ref\ \ NW - vec\ (2,0)\ \text{--}\ cycle]\ (tex\ \texttt{"!"}) \end{array}$$

This example uses a single atomic picture. Often we need references to points in different pictures. This can be done by assigning names to pictures using the attribute function *setName*. A picture can have more than one name.

$$setName \quad :: \quad Name \rightarrow Picture \rightarrow Picture$$

The combination of names (e.g. picture name plus reference point name) is done by the constructor

$$(\triangleleft) \quad :: \quad Name \rightarrow Name \rightarrow Name$$

Names can be character strings or integers as demonstrated in figure 1.9. In the latter case the constructor $\triangleleft$ has to be used, as we will discuss in section 1.16.

Here we see a particularity of paths. The path is directed to the center of the picture but does not reach it. It is cut at the bounding boxes. This is the default behaviour when a path connects different pictures.

$$\begin{array}{l} draw\ [ref\ \ (\texttt{"box1"} \triangleleft C)\ \text{--}\ ref\ \ (2 \triangleleft C)] \\ \quad ((box\ \texttt{"1"}\ \#\ setName\ \texttt{"box1"}) \\ \quad\ \ \square\square\ (box\ \texttt{"2"}\ \#\ setName\ (2 :: Int))) \end{array}$$

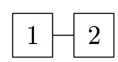


Figure 1.9: A path between two pictures.

The process of naming can also be applied to pictures which are combined from named subpictures. The references have then multipart names. It is allowed to drop parts of the chain of names if the reference is still unique: In the following picture

$$\begin{array}{l} (tex\ \texttt{"a"}\ \#\ setName\ \texttt{"a"})\ \square\square\ (tex\ \texttt{"b"}\ \#\ setName\ \texttt{"b"}) \\ \ \#\ setName\ \texttt{"ab"} \end{array}$$

the expression *ref* ("ab" $\triangleleft$ "a" $\triangleleft$ C) references the center of the picture *tex* "a". The expression *ref* ("a" $\triangleleft$ C) does the same, since only one subpicture of name "a" exist. The reference *ref* ("ab" $\triangleleft$ C) is also allowed but not unique. The rules for the resolution of such ambiguities are explained in section 1.17.

Now we can complete our finite state machine from 1.3. For the drawing of the loops we define a function which takes the name of a picture as argument and adopts the size of the loop to the width of the picture.

$$\begin{array}{lcl} loopN & :: & Name \rightarrow Path \\ loopN\ \ s & = & ref\ \ (s \triangleleft NE)\ ..\ arrow\ (ref\ \ (s \triangleleft N) + vec\ (0, 0.5 * width\ \ s)) \\ & & \qquad\qquad\qquad\qquad (ref\ \ (s \triangleleft NW)) \end{array}$$

$$\begin{array}{lcl} loopSW & :: & Name \rightarrow Path \\ loopSW\ \ s & = & ref\ \ (s \triangleleft SW)\ ..\ arrow\ (ref\ \ (s \triangleleft S) - vec\ (0.353 * width\ \ s, \end{array}$$

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad 0.353 * \mathit{width}\ s))\\
&\qquad\qquad (\mathit{ref}\ (s \triangleleft S))
\end{aligned}
$$

We will discuss arrows thoroughly in section 1.11.

$$
\begin{aligned}
&\mathit{arrow} &&:: \mathit{Point} \to \mathit{Point} \to \mathit{Path}\\
&\mathit{arrow}\ a\ b &&= a \mathrel{..} b \mathbin{\#} \mathit{setArrowHead}\ \mathit{default}
\end{aligned}
$$

The function *width* returns the width of a picture:

$$
\begin{aligned}
&\mathit{width} &&:: \mathit{String} \to \mathit{Numeric}\\
&\mathit{width}\ s &&= \mathit{xpart}\ (\mathit{ref}\ (s \triangleleft E)) - \mathit{xpart}\ (\mathit{ref}\ (s \triangleleft W))
\end{aligned}
$$

This assumes that the point W is the leftmost one and E the rightmost one. Finally we define a function for arrows with labels:

$$
\begin{aligned}
&\mathit{to} &&:: \mathit{Name} \to \mathit{Numeric} \to \mathit{Name} \to \mathit{String} \to \mathit{Dir} \to \mathit{Path}\\
&\mathit{to}\ a\ \mathit{sa}\ b\ l\ d &&= \mathit{arrow}\ (\mathit{ref}\ (a \triangleleft C))\ (\mathit{ref}\ (b \triangleleft C))\\
& && \quad \mathbin{\#} \mathit{setStartAngle}\ \mathit{sa}\\
& && \quad \mathbin{\#} \mathit{setLabel}\ 0.5\ d\ (\mathit{math}\ l)
\end{aligned}
$$

draw [*loopN* "C"
  # *setLabel* 0.5 S (*math* "b"),
  loopSW "B"
  # *setLabel* 0.5 N (*math* "a"),
  to "start" 0 "B" "a" S,
  to "C" (−90) "B" "a" W,
  to "B" 0 "D" "b" S,
  to "D" 0 "stop" "b" S,
  to "start" 55 "C" "b" SE,
  to "stop" 135 "C" "b" SW,
  to "stop" (−125) "B" "a" N,
  to "D" (−145) "B" "a" N
  ] *automat*

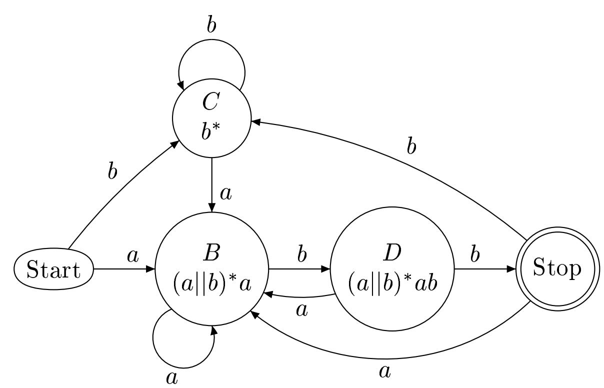


Figure 1.10: Second part of the finite state machine from figure 1.3. The state transitions are now added.

1.6 Numbers and points

Numerical values have the type *Numeric* and points the type *Point* in *functional* METAPOST. Many functions exist to work with variables of these types. We already used the function

$$
\mathit{vec} \quad :: (\mathit{Numeric}, \mathit{Numeric}) \to \mathit{Point}
$$

which generates a point from its coordinates. The inverse operation, extracting a coordinate, is done by

$$
\begin{array}{lcl}
xpart & :: & Point \to Numeric \\
ypart & :: & Point \to Numeric
\end{array}
$$

The type *Point* is instance of the class *Num*. Therefore the basic arithmetic operations $(+)$, $(-)$ and $(*)$ can be applied to points. The type *Numeric* is instance of the classes *Num*, *Fractional*, *Floating* and *Enum*. This allows besides basic arithmetics the application of functions like *sin*, *tan*, *sqrt* or *exp*.

The multiplication of a number with a point is possible by using the operator $(\dot{*})$. The polar coordinates of a point can be calculated using *angle* and $dist\ (vec\ (0,0))$ (giving the distance to the origin).

$$
\begin{array}{lcl}
angle & :: & Point \to Numeric \\
dist & :: & Point \to Point \to Numeric
\end{array}
$$

A point lying on a line between two points can be defined by the function *med* [7] (mediate). The first argument determines where the point will be positioned on the line. For example, $med\ \frac{1}{3}\ z_1\ z_2$ defines the point lying on one third of the line from point z_1 to z_2. *med* can be applied analogously to numbers.

$$
\begin{array}{lcl}
med & :: & Numeric \to Point \to Point \to Point \\
med & :: & Numeric \to Numeric \to Numeric \to Numeric
\end{array}
$$

The largest or smallest member of a list of numbers is found by

$$
\begin{array}{lcl}
maximum' & :: & [Numeric] \to Numeric \\
minimum' & :: & [Numeric] \to Numeric
\end{array}
$$

Assume that we need a circle around point o which has the minimal size to include three points p_1, p_2 and p_3. The radius of this circle can be calculated by

$$
\begin{array}{lcl}
radius & :: & Numeric \\
radius & = & maximum'\ [dist\ o\ p_1, dist\ o\ p_2, dist\ o\ p_3]
\end{array}
$$

Some more useful functions are defined:

$$
\begin{array}{lcl}
dir & :: & Numeric \to Point \\
dir\ a & = & vec\ (cos\ a, sin\ a) \\
 & & \\
xy & :: & Point \to Point \to Point \\
xy\ p_1\ p_2 & = & vec\ (xpart\ p_1, ypart\ p_2) \\
 & & \\
xdist & :: & Point \to Point \to Numeric \\
xdist\ p_1\ p_2 & = & xpart\ p_1 - xpart\ p_2 \\
 & & \\
ydist & :: & Point \to Point \to Numeric \\
ydist\ p_1\ p_2 & = & ypart\ p_1 - ypart\ p_2
\end{array}
$$

[7] The function *med* corresponds to a BERNSTEĬN polynomial of degree one:

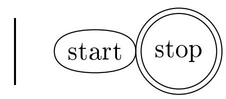
$$med\ t\ z_1\ z_2 = B(z_1, z_2; t) = (1-t)z_1 + tz_2$$

Compare footnote 4 in this chapter.

1.7 Symbolic equations

One highlight of *functional* METAPOST is the possibility to use equations in the definition of pictures. The equations express relations or conditions which shall be fulfilled by the layout.

$$
\begin{array}{l}
\textbf{let } \mathit{beside}\ a\ b = \mathit{overlay}\ [\mathit{ref}\ (0 \triangleleft E) \doteq \mathit{ref}\ (1 \triangleleft W)] \\
\qquad\qquad\qquad\qquad\quad [a, b] \\
\textbf{in } \mathit{beside}\ (\mathit{oval}\ \texttt{"start"})\ (\mathit{circle}\ (\mathit{circle}\ \texttt{"stop"}))
\end{array}
$$



We already know this picture from section 1.3. The function *beside* has the same effect as the $(\square\!\square)$ combinator, here expressed as a condition on the positioning. The function

$$\mathit{overlay} \quad :: \quad [\mathit{Equation}] \to [\mathit{Picture}] \to \mathit{Picture}$$

is used to add equations to the definition of a picture. Here we reference variables and points from picture number $n+1$ by prefixing their name with n, i.e. the first picture has the name 0, the second the name 1 and so on. In our example we state that reference point E of the first point must have the same position as reference point W of the second picture. This fixes the layout.

Another example is an alternative implementation of the function *rowSepBy*, different from the one proposed in section 1.3.

$$
\begin{array}{lcl}
\mathit{rowSepBy} & :: & \mathit{Numeric} \to [\mathit{Picture}] \to \mathit{Picture} \\
\mathit{rowSepBy}\ \mathit{hSep}\ \mathit{ps} & = & \mathit{overlay}\ [\mathit{ref}\ (i \lhd E) + \mathit{vec}\ (\mathit{hSep}, 0) \doteq \mathit{ref}\ (i+1 \lhd W) \\
 & & \qquad\quad \mid i \leftarrow [0\,..\,\mathit{length}\ \mathit{ps} - 2]] \\
 & & \qquad\quad \mathit{ps}
\end{array}
$$

We can define all combinators in this way. Formulating such systems of equation one has to take care that the relative position of each picture is uniquely determined by the equations.

In the previous examples we only stated the equality of several reference points. But it is also possible to define additional variables of type *Point* or type *Numeric*.

$$\mathit{ref}\ \texttt{"point"} :: \mathit{Point}$$

denotes a point variable of the name `"point"`, and

$$\mathit{var}\ \texttt{"number"} :: \mathit{Numeric}$$

a numeric variable of the name `"number"`. Such variables can be used as unknowns the values of which are determined by the equations.

The following equation defines a point variable with name `"target"` which is `"distance"` away from `"point"` in the direction specified by `"angle"`. (In other words, `"distance"` and `"angle"` are the polar coordinates of `"target"` in a coordinate frame centered at `"point"`.)

$$\mathit{ref}\ \texttt{"target"} \doteq \mathit{ref}\ \texttt{"point"} + \mathit{var}\ \texttt{"distance"} \mathbin{\dot{*}} \mathit{dir}\ (\mathit{var}\ \texttt{"angle"})$$

How are new variables defined? The first appearance of a variable with a normal (i.e. not composed by $\triangleleft$ or $\lhd$) name creates a new variable ("defining appearance"). All later appearances of this variables are "applied appearances".

Let us have a closer look at the equations. An equation has the type *Equation* and is generated by the operator

$(\doteq)$ :: $a \to a \to Equation$

The type variable a stands either for *Numeric* or for *Point*; equality is possible only between expressions of the same type. The function

$equal$ :: $[a] \to Equation$

allows to define multiple equalities of the form $x_1 \doteq x_2 \doteq \ldots \doteq x_n$. It is also possible to formulate that an equation should only be imposed if some condition is fulfilled.

$cond$ :: $Boolean \to a \to a \to a$

Hereby the following comparison operators can be applied to points or numbers and define an expression of type *Boolean*.

$(\doteqdot)$ :: $a \to a \to Boolean$
$(\dot{\not\equiv})$ :: $a \to a \to Boolean$
$(\dot{<})$ :: $a \to a \to Boolean$
$(\dot{\leqslant})$ :: $a \to a \to Boolean$

For the *Boolean* type is the usual BOOLEAN algebra implemented, where $(*)$ represents the function And, $(+)$ the function Or and *negate* the function Not.

For a variable number or point whose name is unimportant since it is not used again we can use the special expression *whatever*.The equation

$ref\ z_1 \doteq med\ whatever\ (ref\ z_2)\ (ref\ z_3)$

says that the point z_1 must lie somewhere on the line between the points z_2 and z_3.

But equations can not only define layouts by relating points and coordinates. They can also define variables whose values are used afterwards in a picture. This is possible using the function

$define$ :: $[Equation] \to Picture \to Picture$

The scope of the defined variables is similar to the construct **let**..**in**.. in Haskell. This function can also be applied to paths and areas.

$define$ :: $[Equation] \to Path \to Path$
$define$ :: $[Equation] \to Area \to Area$

We will demonstrate the use of *define* in an example. We want to draw a circle through three given points. This can be done in two steps: find the center of the circumcircle and calculate the radius of the circumcircle. The center of the circumcircle of a triangle is the intersection point of the perpendicular bisectors.

Given are three points with names `"p1"`, `"p2"` and `"p3"`.

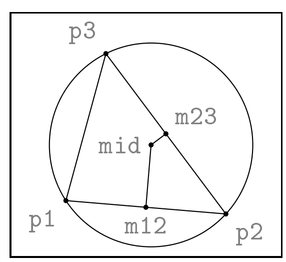


Figure 1.11: Construction of a circumcircle.

$points3$ :: $[Equation]$
$points3$ $=$ $[ref$ `"p1"` $\doteq vec\ (0,5),$
ref `"p2"` $\doteq vec\ (60,0),$
ref `"p3"` $\doteq vec\ (15,60)]$

Now we define the middle points of the lines $\overline{\texttt{p1 p2}}$ and $\overline{\texttt{p2 p3}}$.

$$
\begin{aligned}
\mathit{meds3} \quad &= [\mathit{ref}\ \texttt{"m12"} \doteq \mathit{med}\ 0.5\ (\mathit{ref}\ \texttt{"p1"})\ (\mathit{ref}\ \texttt{"p2"}),\\
&\mathit{ref}\ \texttt{"m23"} \doteq \mathit{med}\ 0.5\ (\mathit{ref}\ \texttt{"p2"})\ (\mathit{ref}\ \texttt{"p3"})]
\end{aligned}
$$

The angle between these lines and the perpendiculars:

$$
\begin{aligned}
\mathit{angles3} \quad &= [\mathit{var}\ \texttt{"a12"} \doteq 90 + \mathit{angle}\ (\mathit{ref}\ \texttt{"p1"} - \mathit{ref}\ \texttt{"p2"}),\\
&\mathit{var}\ \texttt{"a23"} \doteq 90 + \mathit{angle}\ (\mathit{ref}\ \texttt{"p2"} - \mathit{ref}\ \texttt{"p3"})]
\end{aligned}
$$

The circle center *ref* `"mid"` is somewhere on the line through *ref* `"m12"` with angle *var* `"a12"` and also on the line through *ref* `"m23"` with angle *var* `"a23"`.

$$
\begin{aligned}
\mathit{mid3} \quad &= [\mathit{equal}\ [\mathit{ref}\ \texttt{"mid"},\\
&\qquad\quad \mathit{med}\ \mathit{whatever}\ (\mathit{ref}\ \texttt{"m12"})\\
&\qquad\qquad\qquad\qquad (\mathit{ref}\ \texttt{"m12"} + \mathit{dir}\ (\mathit{var}\ \texttt{"a12"})),\\
&\qquad\quad \mathit{med}\ \mathit{whatever}\ (\mathit{ref}\ \texttt{"m23"})\\
&\qquad\qquad\qquad\qquad (\mathit{ref}\ \texttt{"m23"} + \mathit{dir}\ (\mathit{var}\ \texttt{"a23"}))]]
\end{aligned}
$$

The radius is the distance of the center *ref* `"mid"` from one of the original points.

$$
r_3 \quad = [\mathit{var}\ \texttt{"r"} \doteq \mathit{dist}\ (\mathit{ref}\ \texttt{"mid"})\ (\mathit{ref}\ \texttt{"p1"})]
$$

Now we have everything necessary to draw the circumcircle. For clarity we also draw the triangle and the two perpendicular bisectors. The lists of equations are merged with the (&) operator.

$$
\begin{aligned}
&\mathit{define}\ (\mathit{points3}\ \&\ \mathit{meds3}\ \&\ \mathit{angles3}\ \&\ \mathit{mid3}\ \&\ r_3)\\
&\qquad (\mathit{draw}\ [\mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (0, \mathit{var}\ \texttt{"r"})\ ..\ \mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (\mathit{var}\ \texttt{"r"}, 0)\\
&\qquad\qquad\quad ..\ \mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (0, -\mathit{var}\ \texttt{"r"})\ ..\ \mathit{ref}\ \texttt{"mid"} + \mathit{vec}\ (-\mathit{var}\ \texttt{"r"}, 0)\\
&\qquad\qquad\quad ..\ \mathit{cycle},\\
&\qquad\qquad\quad \mathit{ref}\ \texttt{"p1"}\ \text{--}\ \mathit{ref}\ \texttt{"p2"}\ \text{--}\ \mathit{ref}\ \texttt{"p3"}\ \text{--}\ \mathit{cycle},\\
&\qquad\qquad\quad \mathit{ref}\ \texttt{"m12"}\ \text{--}\ \mathit{ref}\ \texttt{"mid"},\\
&\qquad\qquad\quad \mathit{ref}\ \texttt{"m23"}\ \text{--}\ \mathit{ref}\ \texttt{"mid"}]\ \mathit{empty})
\end{aligned}
$$

1.8 Colors

Every visible object can have a color. The default is *black*. The color space uses the RGB model, i.e. a color is given by the additive mixing of red, green and blue.

$$
\mathit{color} \quad :: \mathit{Double} \to \mathit{Double} \to \mathit{Double} \to \mathit{Color}
$$

Some often used colors have predefined names:

$$
\begin{aligned}
\mathit{white} \quad &= \mathit{color}\ 1\ 1\ 1\\
\mathit{black} \quad &= \mathit{color}\ 0\ 0\ 0\\
\mathit{red} \quad &= \mathit{color}\ 1\ 0\ 0\\
\mathit{green} \quad &= \mathit{color}\ 0\ 1\ 0\\
\mathit{blue} \quad &= \mathit{color}\ 0\ 0\ 1\\
\mathit{yellow} \quad &= \mathit{color}\ 1\ 1\ 0\\
\mathit{cyan} \quad &= \mathit{color}\ 0\ 1\ 1\\
\mathit{magenta} \quad &= \mathit{color}\ 1\ 0\ 1\\
\mathit{grey}\ n \quad &= \mathit{color}\ n\ n\ n
\end{aligned}
$$

The type *Color* is an instance of *Num* and *Fractional*. So we can add, subtract and multiply colors. These operations are defined componentwise. Therefore, *red* + *green* gives the same color as *yellow* and *cyan* − *blue* is the same as *green*. The function *fromRational* is implemented, too. The number 0.5 is interpreted as *grey* 0.5, i.e. *color* 0.5 0.5 0.5. The values for color percentages should be in the interval $[0;1]$. Every number outside this interval is interpreted as either 0 or 1.

The color of a picture is changed by

$$setColor \quad :: \quad Color \to Picture \to Picture$$

The background color is changed by

$$setBGColor \quad :: \quad Color \to Picture \to Picture$$

```
tex "colors"
# setColor green
# setBGColor 0.2
```

Figure 1.12: Color attributes

Figure 1.12 shows how to apply the color attribute functions.

1.9 Dash patterns

Paths and frames can be drawn with different dash patterns. The lengths of the pieces which are alternatingly drawn or not drawn can be given as a list.[8]

$$dashPattern \quad :: \quad [Double] \to Pattern$$
$$dashPattern' \quad :: \quad [Double] \to Pattern$$

The difference between these functions is that *dashPattern* starts with a drawn piece, *dashPattern'* with an empty piece. Of course, the pattern is repeated for longer paths.

Some patterns are already predefined:

$$dashed \quad :: \quad Pattern$$
$$dashed \quad = \quad dashPattern\ [3,3]$$

$$dotted \quad :: \quad Pattern$$
$$dotted \quad = \quad dashPattern'\ [2.5,0,2.5]$$

A path segment or a frame gets a pattern by the attribute function

$$setPattern \quad :: \quad Pattern \to Path \to Path$$

as can be seen in figure 1.13.

[8] There is a limit: PostScript allows no more than eleven entries in this list.

1.10 Pencils

Another attribute of paths and frames are pencils. They come in two sorts, rectangular and oval and they can be rotated.

$$
\begin{array}{lcl}
\mathit{penSquare} & :: & (\mathit{Numeric}, \mathit{Numeric}) \to \mathit{Numeric} \to \mathit{Pen} \\
\mathit{penCircle} & :: & (\mathit{Numeric}, \mathit{Numeric}) \to \mathit{Numeric} \to \mathit{Pen}
\end{array}
$$

The first parameter sets the size, the second the rotation angle. This allows calligraphic effects.

```
let path = vec (0,0) .. vec (10,10)
             .. vec (30,−10) .. vec (40,0)
in column [path # setPattern dotted,
             path # setPattern dashed,
             path # setPattern (dashPattern [0,2,3,2])
                  # setPen 1,
             path # setPen (penCircle (0.1,5) 20),
             path # setPen (penSquare (5,5) 45),
             path # setPen 5]
```

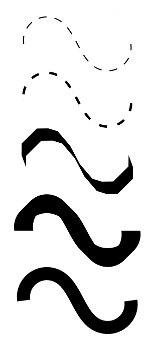


Figure 1.13: Dash patterns and calligraphic effects

A circular pen of given size is perhaps more frequently used. The command *setPen* 1.5 selects a circular pen of diameter 1.5 PostScript points.

1.11 Arrows

Path segments can have arrow heads on both ends. Besides the common *defaultArrowHead*[9] it is possible to prescribe the length of the arrowhead and the opening angle. An example is shown in figure 1.14.

$$
\begin{array}{lcl}
\mathit{arrowHeadSize} & :: & \mathit{Double} \to \mathit{Double} \to \mathit{PathArrowHead} \\
\mathit{defaultArrowHead} & :: & \mathit{PathArrowHead} \\
\mathit{default} & = & \mathit{defaultArrowHead}
\end{array}
$$

A bigger arrowhead is predefined:

$$
\begin{array}{lcl}
\mathit{arrowHeadBig} & :: & \mathit{PathArrowHead} \\
\mathit{arrowHeadBig} & = & \mathit{pathArrowHeadSize}\ 8\ 4
\end{array}
$$

An arrowhead comes in two different styles, a filled triangle or a cusp of two lines:

[9] See Appendix A for the *default* keyword!

$$
\begin{aligned}
&\mathit{ahFilled} &&:: \mathit{ArrowHeadStyle}\\
&\mathit{ahLine} &&:: \mathit{ArrowHeadStyle}
\end{aligned}
$$

The style can be changed or read by attribute functions:

$$
\begin{aligned}
&\mathit{setArrowHeadStyle} &&:: \mathit{ArrowHeadStyle} \to \mathit{ArrowHead} \to \mathit{ArrowHead}\\
&\mathit{getArrowHeadStyle} &&:: \mathit{ArrowHead} \to \mathit{ArrowHeadStyle}
\end{aligned}
$$

The following functions add arrowheads to the end or start of a path segment or return an arrowhead.

$$
\begin{aligned}
&\mathit{setArrowHead} &&:: \mathit{ArrowHead} \to a \to a\\
&\mathit{setStartArrowHead} &&:: \mathit{ArrowHead} \to a \to a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{getArrowHead} &&:: a \to \mathit{Maybe\ ArrowHead}\\
&\mathit{getStartArrowHead} &&:: a \to \mathit{Maybe\ ArrowHead}
\end{aligned}
$$

There is a special function for the simplest arrows:

$$\mathit{arrow}\ a\ b \quad = \quad a\ ..\ b\ \#\ \mathit{setArrowHead\ default}$$

```
let { ar2 a = setArrowHead a ∘ setStartArrowHead a;
      lineStyle = setArrowHeadStyle ahLine;
      fs = [id,
            ar2 defaultArrowHead,
            ar2 (arrowHeadSize 10 20),
            ar2 (arrowHeadSize 5 250),
            ar2 (defaultArrowHead # lineStyle),
            ar2 (arrowHeadSize 10 20 # lineStyle),
            ar2 (arrowHeadSize 5 180 # lineStyle),
            ar2 (arrowHeadSize 5 250 # lineStyle)]; }
in [f (vec (0, y) -- vec (40, y))
   | (y, f) ← zip [0, −16 ..] fs]
```

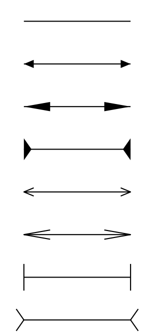


Figure 1.14: Different sorts of arrows. Look at the fourth and eighth arrow: If the opening angle is larger than 180 the head points backwards. For the filled triangle variant it is drawn in such a way that the overall length of the arrow does not change.

1.12 Areas

Besides the possibility to draw lines a universal graphics language should have the possibility to fill areas. Areas are, like paths, independent objects with their own attributes like color, pen and drawing order. Different from paths these attributes exist only once for the whole area and not individually for each path segment. An area can be created from a cyclic path:

toArea :: *Path* → *Area*

The attribute functions to set color or pen can be applied to the area object. A new attribute is the drawing order: by default, the area conceals everything under it. But it is also possible to draw an area behind a picture:

setBack :: *Area* → *Area*
setFront :: *Area* → *Area*

```
let p = ref (0 ◃ C) -- ref (1 ◃ C)
        -- ref (2 ◃ C) -- ref (3 ◃ C)
        -- ref (2 ◃ C) -- ref (1 ◃ C) -- cycle
in fill [toArea p # setBack # setPen 13,
         toArea p # setBack # setPen 12 # setColor 0.9]
        binTree
```

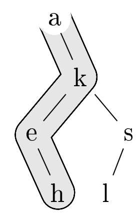


Figure 1.15: Areas can be drawn behind pictures using the attribute function # *setBack*.

The function

fill :: [*Area*] → *Picture* → *Picture*

adds areas to a picture. Figure 1.15 shows how useful it can be to use *setBack* to draw areas behind pictures. For the resulting bounding box applies basically the same as in the case of paths: the adding of areas does not change the bounding box of a picture.

```
let p = vec (−1 ∗ cm, 0)
        .. vec (0, −1 ∗ cm)
        .. vec (1 ∗ cm, 0)
        # setStartVector down
        # setEndVector up
in
fill [(p .. vec (0, 0)
         # setEndVector (vec (−1, −2)))
       .. cycle
        # setEndVector up]
(p .. vec (0, 1 ∗ cm) .. cycle)
```

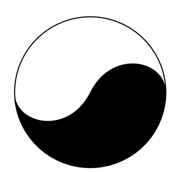


Figure 1.16: Arbitrary cyclic paths can be filled. (compare Figure 21 in [Hob92]).

1.13 Clipping

One effect of METAPOST, also realized in *functional* METAPOST, is the possibility to cut a picture along a cyclic path. Figure 1.17 shows what this means. Everything outside the path vanishes. The bounding box of the resulting picture is the minimal enclosing rectangle.

clip :: *Path* → *Picture* → *Picture*

Only the form of the given path is important. Other attributes like color, dash pattern, pens or labels of the path are ignored.

```
clip (ref SE + vec (0,15) -- ref C
    --ref NE − vec (0,15) .. ref W
    ..cycle)
    (matrixSepBy 0 0
            (replicate 8 (replicate 4 "pac")))
```

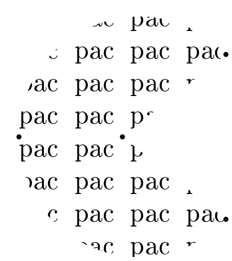


Figure 1.17: The picture is cut along the given path. Four points along the path are emphasized for clarity.

1.14 Transformations

Often we want to change the size of a picture or to rotate some text. This is possible by affine transformations which can be applied to arbitrary pictures.[10] Predefined are e.g. commands to scale, rotate, skew and reflect pictures.

scale :: *Numeric* → *Picture* → *Picture*
scaleX :: *Numeric* → *Picture* → *Picture*
scaleY :: *Numeric* → *Picture* → *Picture*
rotate :: *Numeric* → *Picture* → *Picture*
skewX :: *Numeric* → *Picture* → *Picture*
skewY :: *Numeric* → *Picture* → *Picture*
reflectX :: *Picture* → *Picture*
reflectY :: *Picture* → *Picture*

Again, the resulting bounding box is the minimal enclosing rectangle.

When referencing a named point in a transformed picture, the transformation is automatically taken into account. In this way, e.g., a line defined using that point really touches the point and not the untransformed position.

[10] Since lines are always areas in PostScript, this may change the linewidth.

rotate 45 *color1*
□□ *scale* 2 (*rotate* 45 *color1*)
□□ *reflectY* (*skewX* 1.2 *color1*)
□□ *rotate* 45 (*skewY* 1.2 *color1*)
□□ *skewY* 1.2 (*rotate* 45 *color1*)

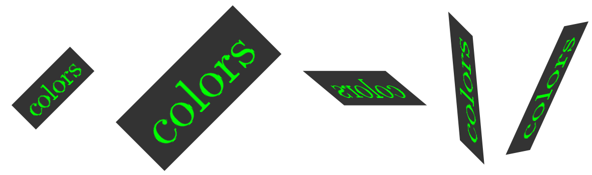


Figure 1.18: Different affine transformations.

1.15 Bitmap graphics

Unfortunately, METAPOST does not support bitmaps. Nonetheless, by a trick,[11] they are implemented in *functional* METAPOST.

Pictures in black&white (one bit per point), eight bit gray values or 24 bit color values are possible. A point has the edge length of $\frac{1}{600}$ inch. Figure 1.19 shows a bitmap with one bit depth, magnified by a factor of twenty.

```
let turkey = ["003B00","002700","002480","0E4940",
              "114920","14B220","3CB650","75FE88",
              "17FF8C","175F14","1C07E2","3803C4",
              "703182","F8EDFC","B2BBC2","BB6F84",
              "31BFC2","18EA3C","0E3E00","07FC00",
              "03F800","1E1800","1FF800"]
in scale 20 (image Depth1 turkey)
```

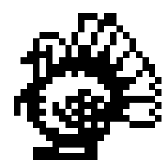


Figure 1.19: A bitmap, compare [Ado85].

1.16 Subtypes of *Picture*, *Path* and *Name*

So far we simplified things a bit by assuming that all pictures have the type *picture*. But reality is a bit more complicated:

1. Frames are special pictures. They have additional attributes (like the distance to the framed picture) and corresponding attribute functions (like *setDX*). Some attributes have a different meaning. E.g., *setColor* sets the color of the frame, not of the rest of the picture.
   Therefore frames are not objects of type *Picture* but of type *Frame*.

[11] This trick needs the METAFONT fonts fmp1, fmp8 and fmp24 distributed with *functional* METAPOST.

Conceptually, *Frame* should be a subtype of *Picture* with some additional attributes. But Haskell does not support subtypes.[12]

2. Essentially the same situation applies to paths and names: we want subtypes with special properties.
3. We want to define special types for special purposes (canvas graphics, trees). It should be possible to convert these objects into pictures.

Haskell does not allow subtypes, but it has the concept of (type) classes. We will now explain how *functional* METAPOST uses this feature to solve the problems mentioned above.

1.16.1 *Picture*

The trick is to create a class *IsPicture* and to make all "subtypes of *Picture*" instances of this class.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class}\ (\mathit{Show}\ a) \Rightarrow \mathit{IsPicture}\ a\ \textbf{where}} \\
\quad \mathit{toPicture} & :: & a \to \mathit{Picture} \\
\quad \mathit{toPictureList} & :: & [a] \to \mathit{Picture} \\
\quad \mathit{toPicture}\ a & = & \mathit{text}\ (\mathit{show}\ a) \\
\quad \mathit{toPictureList}\ \mathit{ps} & = & \mathit{row}\ (\mathit{map}\ \mathit{toPicture}\ \mathit{ps})
\end{array}
$$

The function *toPicture* is the identity map for pictures. Other types which shall be subclasses of *Picture* must be instances of *IsPicture* and must implement the *toPicture* function. This way, expressions of a subtype can get converted to a picture.

The *toPicture* function is implicitly used by all functions which actually expect arguments of *Picture* type, so that they accept all subtypes (i.e. instances of *IsPicture*), too.

The type of the ($\square\square$) combinator is therefore really

$$
\begin{array}{lll}
(\square\square) & :: & (\mathit{IsPicture}\ a, \mathit{IsPicture}\ b) \Rightarrow a \to b \to \mathit{Picture} \\
p_1 \mathbin{\square\square} p_2 & = & \mathit{row}\ [\mathit{toPicture}\ p_1, \mathit{toPicture}\ p_2]
\end{array}
$$

The user can define his own instances. Let us give an example: we define a data type in order to draw trees.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Tree} & = & N\ \mathit{Tree}\ \mathit{Picture}\ \mathit{Tree} \\
 & \mid & E
\end{array}
$$

In order to make *Tree* to a subtype of *Picture* we need an instance declaration in the following form:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{IsPicture}\ \mathit{Tree}\ \textbf{where}} \\
\quad \mathit{toPicture}\ t & = & \mathit{draw}\ \mathit{edges}\ (\mathit{overlay}\ \mathit{equations}\ \mathit{nodePics}) \\
\qquad\qquad \textbf{where} & & \\
\qquad\qquad \mathit{edges} & = & \ldots \\
\qquad\qquad \mathit{equations} & = & \ldots \\
\qquad\qquad \mathit{nodePics} & = & \ldots
\end{array}
$$

[12] In C++ one could define a class *Picture* with a virtual function *setColor* and make *Frame* a derived class with additional member function *setDX*.

Now we can use expressions of type *Tree* just as normal pictures:

$$
\begin{array}{lll}
\mathit{pic} & :: & \mathit{Picture} \\
\mathit{pic} & = & t_1 \,\Box\Box\, t_2 \\
\quad\quad \textbf{where} & & \\
\quad\quad t_1, t_2 & :: & \mathit{Tree} \\
\quad\quad t_1 & = & N\ E\ (\mathit{tex}\ \texttt{"2"})\ (N\ E\ (\mathit{tex}\ \texttt{"3"})\ E) \\
\quad\quad t_2 & = & N\ (N\ E\ (\mathit{tex}\ \texttt{"1"})\ E)\ (\mathit{tex}\ \texttt{"2"})\ (N\ E\ (\mathit{tex}\ \texttt{"3"})\ E)
\end{array}
$$

Functional METAPOST already makes many types to instances of the *IsPicture* class, including the types *Char*, *String*, *Int*, *Integer*, *Numeric* as well as tuples, triples and lists of them.

$$
\mathit{pic}' \quad = \quad \texttt{"String"} \,\Box\Box\, 2
$$

unfortunately, Haskell lists must have homogeneous types, therefore expressions like $[\texttt{"String"}, 2]$ are not allowed.

The types *Path* and *Area* are also instances of the *IsPicture* class. At the end of section 1.4 we showed one way to convert a path into a picture. Now we have learned an easier way: we can apply all functions which expect picture arguments directly to paths, too.

1.16.2 *Path*

A similar problem arises for path constructors. They have not really the type $\mathit{Path} \to \mathit{Path} \to \mathit{Path}$, otherwise we could not connect paths **and** points by path constructors.

In reality, the arguments of path constructors can have arbitrary types which are instances of the *IsPath* class and therefore have a *toPath* function implemented.

$$
\begin{array}{lll}
\textbf{class}\ \mathit{IsPath}\ a\ \textbf{where} & & \\
\quad \mathit{toPath} & :: & a \to \mathit{Path} \\
\quad \mathit{toPathList} & :: & [a] \to \mathit{Path} \\
\quad \mathit{toPathList}\ \mathit{ps} & = & \mathit{foldl1}\ (\text{--})\ (\mathit{map}\ \mathit{toPath}\ \mathit{ps})
\end{array}
$$

So, the type of the path constructors really is

$$
(\&), (..), (...), (\text{--}), (\text{---}) \quad :: \quad (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \to b \to \mathit{Path}
$$

As examples we list some instance declarations of important types which allow an intuitive and short notation for paths.

$$
\begin{array}{lll}
\textbf{instance}\ \mathit{IsPath}\ \mathit{Path}\ \textbf{where} & & \\
\quad \mathit{toPath} & = & \mathit{id} \\
 & & \\
\textbf{instance}\ \mathit{IsPath}\ \mathit{Point}\ \textbf{where} & & \\
\quad \mathit{toPath} & = & \mathit{PathPoint} \\
 & & \\
\textbf{instance}\ \mathit{IsPath}\ \mathit{Name}\ \textbf{where} & & \\
\quad \mathit{toPath} & = & \mathit{toPath} \circ \mathit{ref}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{instance } \mathit{IsPath}\ a \Rightarrow \mathit{IsPath}\ [\,a\,] \textbf{ where} & & \\
\quad \mathit{toPath} & = & \mathit{toPathList}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{instance } (\mathit{Num}\ a, \mathit{Num}\ b, \mathit{Real}\ a, \mathit{Real}\ b) \Rightarrow \mathit{IsPath}\ (a, b) \textbf{ where} & & \\
\quad \mathit{toPath}\ (a, b) & = & \mathit{toPath}\ (\mathit{vec}\ (\mathit{fromRational}\ \$\ \mathit{toRational}\ a, \\
 & & \qquad \mathit{fromRational}\ \$\ \mathit{toRational}\ b))
\end{array}
$$

This allows not only to connect points by path connectors but also to abbreviate an expression like $\mathit{ref}\ n_1$ -- $\mathit{ref}\ n_2$ to n_1 -- n_2 or even $[n_1, n_2]$. The last declaration allows us to write expressions like $(0,0)$ -- $(10,0)$ -- $(10,10)$ -- *cycle*. This is maximal convergence to the syntax of METAPOST.

1.16.3 *Name*

Names can be constructed from parts of the types *Int*, *String* or *Dir*. Again, we declare them all as instances of a *IsName* class.

Integer constants need a special treatment. Expressions of the type $1 \triangleleft C$ do not work since the type of a numerical constant is not *Int* in Haskell. One can add an explicit type signature and write $(1 :: \mathit{Int}) \triangleleft C$ or one can use the additional function

$$
\begin{array}{lcl}
(\blacktriangleleft) & :: & (\mathit{IsName}\ a) \Rightarrow \mathit{Int} \rightarrow a \rightarrow \mathit{Name} \\
(\blacktriangleleft) & = & (\triangleleft)
\end{array}
$$

which fixes the type of the first argument and write $1 \blacktriangleleft C$. Figure 1.20 gives an overview of the system of subtypes of *functional* METAPOST.

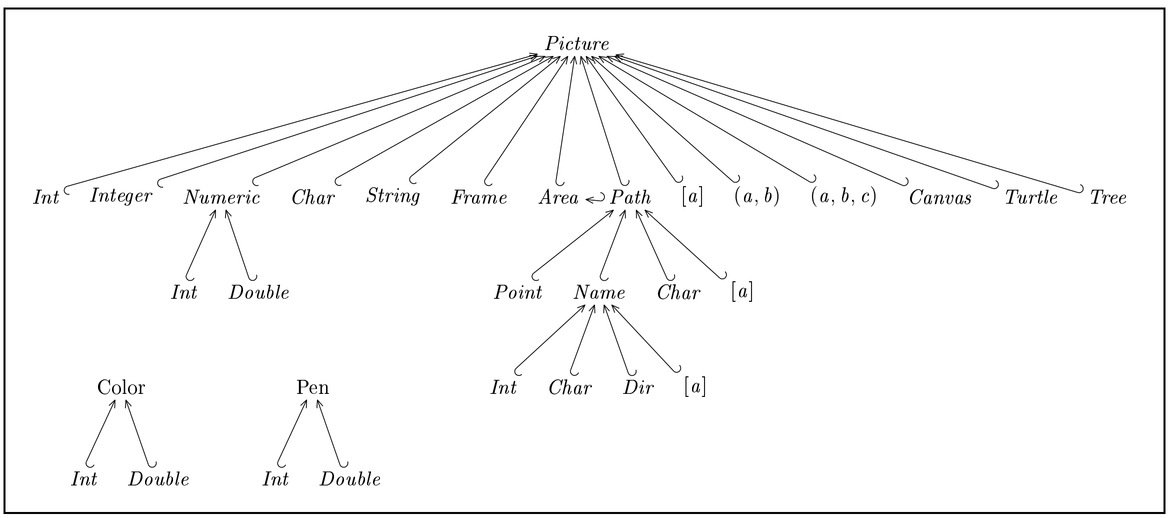


Figure 1.20: Chart of all subtypes. The relation $[\,a\,] \hookrightarrow \mathit{Picture}$ holds for all types a for which $a \hookrightarrow \mathit{Picture}$ holds.

1.17 Visibility and hiding of variables

Variables which have their defining appearance in the *equations* part of an expression of the form *define equations picture* are visible only in this *picture* part.

Variables defined in the *equations* part of an expression of the form *overlay equations pictures* are globally visible.

The names of variables must not be unique. An example are the names of the reference points $C, ..., NW$ which appear in every picture. Therefore we need rules for the hiding of variables to define unambiguously which variable is referenced by an applied appearance.

In the following we abbreviate "defining appearance" by DA and "applied appearance" by AA An AA of a variable references that DA of this variable which is not hidden at this place.

We also need the concept of a global DA: A global DA is one which is not in the same expression as the AA. E.g., for the expression p_1 from the expression *overlay eqs* $[p_1, p_2]$ are all DAs in p_2 global DAs. But the DAs of the equations *eqs* and of the picture p are not global within the expression *define eqs p*. With other words, global DAs are DAs which come from the outer context of an expression. Now we can formulate the hiding rules.

1. A global DA is hidden by a non global one.
2. In the expression *overlay eqs ps*, DAs from *eqs* hide all DAs from the pictures *ps*.
3. For an expression p in the context *define eqs p* hide the global DAs from *eqs* all other global DAs.
4. For two DAs in the pictures p_i and $p_j, 1 \leqslant i, j \leqslant n$ of the expression *overlay eqs* $[p_1 \mathinner{..} p_n]$ we have: if $i < j$ then a DA in p_i hides a DA in p_j.
5. Within a system of equations hides a DA of a variable all other DAs of this variable outside this system.

This definition of hiding rules has been chosen with care. Let us imagine what could happen without rule 1: global DAs would hide DAs in an expression. Then the layout of a picture would be context dependent. This is not acceptable since we consider pictures as reusable building blocks.

The rules 2 and 4 formalize the idea that a "nearer" DA hides a "distant" DA. Rule 4 defines the hiding order for DAs from arguments of *overlay*. This is enough since all picture combinators are defined in terms of *overlay*. Finally, rule 5 ensures that DAs in a system of equation are always used by the AAs within this system. This can be considered as a special case of rule 2.

Chapter 2

Extensions of *functional* METAPOST

The first chapter gave an introduction into the *functional* METAPOST language. It used a special concept in order to describe pictures: Subpictures are aligned relative to each other and thereby combined into new pictures. It is possible to add paths and areas to pictures. This concept is quite good for the description of e.g. charts and plots. But sometimes another concept is more advantageous.

The core language of *functional* METAPOST is efficient enough to allow the implementation of extensions realizing different concepts.

One such extension is canvas graphics, using a concept shared by many graphical interfaces: A picture is drawn by a sequence of single drawing commands.

Another one is turtle graphics: A virtual pen (the "turtle") is navigated by command sequences.

Finally we give an example of a more complex extension, trees with automatically generated layout.

All those extensions can be combined and imbedded into each other.

2.1 Canvas graphics

It is often useful to describe pictures by a local system of coordinates. This concept uses the metaphor of a canvas to which drawing operations are applied.

The drawing command draws a path on a *Canvas*.[1]

$$\mathit{cdraw} \quad :: \quad \mathit{Path} \to \mathit{Canvas}$$

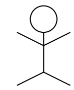
Drawing commands are composed by the sequence operator

$$(\&) \quad :: \quad \mathit{Canvas} \to \mathit{Canvas} \to \mathit{Canvas}$$

This allows already to doodle a bit:

[1]We continue to write type annotations in a simplified form which is easier to read than $\mathit{cdraw} :: \mathit{IsPath}\ a \Rightarrow a \to \mathit{Canvas}$.

$$
\begin{array}{ll}
 & \mathit{cdraw}\ (\mathit{vec}\ (0,5)\ \text{--}\ \mathit{vec}\ (0,-10)) \\
\& & \mathit{cdraw}\ (\mathit{vec}\ (10,5)\ \text{--}\ \mathit{vec}\ (0,0)\ \text{--}\ \mathit{vec}\ (-10,5)) \\
\& & \mathit{cdraw}\ (\mathit{vec}\ (10,-15)\ \text{--}\ \mathit{vec}\ (0,-10)\ \text{--}\ \mathit{vec}\ (-10,-15)) \\
\& & \mathit{cdraw}\ (\mathit{vec}\ (0,5)\ ..\ \mathit{vec}\ (-5,10)\ ..\ \mathit{vec}\ (0,15)\ ..\ \mathit{vec}\ (5,10)\ ..\ \mathit{cycle})
\end{array}
$$



There are further commands to draw a list of paths, to fill areas and to clip. The *Canvas* type is instance of the *IsPicture* class and therefore a canvas graphics can be used like a picture as argument of combinators and other functions.

$$
\begin{array}{lll}
\mathit{cdraws} & :: & [\mathit{Path}] \rightarrow \mathit{Canvas} \\
\mathit{cfill} & :: & \mathit{Area} \rightarrow \mathit{Canvas} \\
\mathit{cfills} & :: & [\mathit{Area}] \rightarrow \mathit{Canvas} \\
\mathit{cclip} & :: & \mathit{Path} \rightarrow \mathit{Canvas} \\
\mathit{relax} & :: & \mathit{Canvas}
\end{array}
$$

A picture can be embedded at an arbitrary position into a canvas.

$$
\begin{array}{lll}
\mathit{cdrop} & :: & (\mathit{Numeric}, \mathit{Numeric}) \rightarrow \mathit{Path} \rightarrow \mathit{Canvas}
\end{array}
$$

Figure 2.1 shows another typical application.

All coordinate data in a sequence of drawing commands refer to the same local system of coordinates. But after conversion of the canvas graphics to a picture it can be placed freely besides other pictures.

$$
\begin{array}{ll}
 & \mathit{cdraws}\ [\mathit{arrow}\ (\mathit{vec}\ (-10,0))\ (\mathit{vec}\ (210,0)), \\
 & \qquad\quad \mathit{arrow}\ (\mathit{vec}\ (0,-50))\ (\mathit{vec}\ (0,50))] \\
\& & \mathit{cdrop}\ (-6,50)\ (\mathit{math}\ \texttt{"y"}) \\
\& & \mathit{cdrop}\ (210,-6)\ (\mathit{math}\ \texttt{"x"}) \\
\& & \mathit{cdraw}\ (\mathit{toPath}\ [\mathit{vec}\ (i, 40 * \mathit{sin}\ (3 * i)) \mid i \leftarrow [0,2\,..\,200]]) \\
\& & \mathit{cdraw}\ (\mathit{toPath}\ [\mathit{vec}\ (i, 40 * \mathit{sin}\ (3 * i) ** 2) \mid i \leftarrow [0,2\,..\,200]])
\end{array}
$$

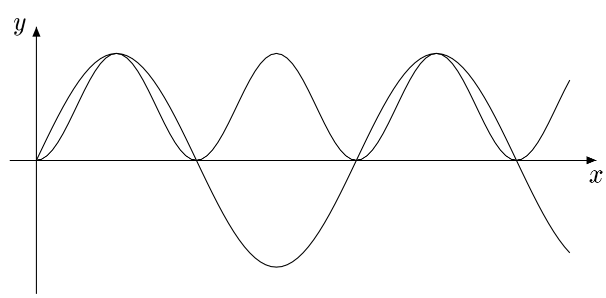


Figure 2.1: A function plot using canvas graphics.

2.2 Turtle graphics

Turtle graphics [Ad82] is the name of the drawing concept of the language LOGO [Abe85]. Pictures arise by the movements of a virtual pen, the turtle[2] The turtle has two states "up" and "down". Movements of the pen in the up state leave a line on the screen. Rotation commands change the direction of the turtles.

The two basic commands for driving the turtle move it forward or rotate it:

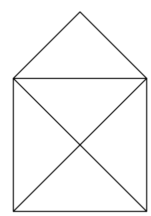
$$\begin{array}{lcl} \mathit{forward} & :: & \mathit{Numeric} \to \mathit{Turtle} \\ \mathit{turn} & :: & \mathit{Numeric} \to \mathit{Turtle} \end{array}$$

Together with the sequence operator

$$\begin{array}{lcl} (\&) & :: & \mathit{Turtle} \to \mathit{Turtle} \to \mathit{Turtle} \end{array}$$

this allows the description of a turtle path.

$$\begin{array}{ll} & \mathit{forward}\ 50\ \&\ \mathit{turn}\ 90 \\ \& & \mathit{forward}\ 50\ \&\ \mathit{turn}\ 90 \\ \& & \mathit{forward}\ 50\ \&\ \mathit{turn}\ 90 \\ \& & \mathit{forward}\ 50\ \&\ \mathit{turn}\ 135 \\ \& & \mathit{forward}\ (50 * \mathit{sqrt}\ 2)\ \&\ \mathit{turn}\ 90 \\ \& & \mathit{forward}\ (25 * \mathit{sqrt}\ 2)\ \&\ \mathit{turn}\ 90 \\ \& & \mathit{forward}\ (25 * \mathit{sqrt}\ 2)\ \&\ \mathit{turn}\ 90 \\ \& & \mathit{forward}\ (50 * \mathit{sqrt}\ 2) \end{array}$$



Besides we have commands to lower and raise the pen, to go back to the starting point and a "do nothing" command which is useful to convert a command list to a turtle path by the function *foldr* (&) *relax*.

$$\begin{array}{lcl} \mathit{penUp} & :: & \mathit{Turtle} \\ \mathit{penDown} & :: & \mathit{Turtle} \\ \mathit{home} & :: & \mathit{Turtle} \\ \mathit{relax} & :: & \mathit{Turtle} \end{array}$$

Often we want to turn by 90 degree.

$$\begin{array}{lcl} \mathit{toleft}, \mathit{toright} & :: & \mathit{Turtle} \\ \mathit{toleft} & = & \mathit{turn}\ 90.0 \\ \mathit{toright} & = & \mathit{turn}\ (-90.0) \end{array}$$

A nice application for turtle graphics are fractal lines, which can be described especially easily (see figure 2.2).

According to our type concept the *turtle* type, too, is an instance of *IsPicture*. The embedding of an arbitrary picture into a turtle graphics is possible by

$$\begin{array}{lcl} \mathit{fromPicture} & :: & \mathit{Picture} \to \mathit{Turtle} \end{array}$$

[2]The name "turtle" was meant literally: The language LOGO was created by Seymour Papert [Pap82] in order to introduce children to the world of programming. The pen on the screen has the form of a turtle.

$$
\begin{array}{lll}
\textit{dragon}\ g\ l & & \\
\quad \mid g \equiv 0 & = & \textit{forward}\ l \mathbin{\#} \textit{setPen}\ 1 \\
\quad \mid g > 0 & = & \textit{dragon}\ (g-1)\ l \\
 & & \&\textit{toright} \\
 & & \&\textit{dragon}\ (-(g-1))\ l \\
\quad \mid \textit{otherwise} & = & \textit{dragon}\ (-(g+1))\ l \\
 & & \&\textit{toleft} \\
 & & \&\textit{dragon}\ (g+1)\ l
\end{array}
$$

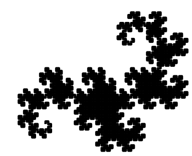


Figure 2.2: Filled dragon line of depth eleven. The pencil width corresponds to the segment length. This creates the filled dragon line. An extensive discussion of the dragon line can be found in [Man87], chapter 7.

which inserts the picture at the current turtle position.

An addition, normally not found in turtle graphics, is the *fork* command which creates a second turtle path at the current turtle position. This allows to draw tree like structures as in figure 2.3.

$$
\textit{fork} \quad :: \quad \textit{Turtle} \rightarrow \textit{Turtle} \rightarrow \textit{Turtle}
$$

Finally an example where we use a function as argument. The following function draws a red roof of width l.

$$
\begin{array}{lll}
\textit{roof} & :: & \textit{Numeric} \rightarrow \textit{Turtle} \\
\textit{roof}\ l & = & \textit{turn}\ 45 \mathbin{\&} \textit{forward}\ (0.5 * l * \textit{sqrt}\ 2) \mathbin{\&} \textit{toright} \\
 & & \&\textit{forward}\ (0.5 * l * \textit{sqrt}\ 2) \mathbin{\&} \textit{turn}\ (-45) \\
 & & \#\ \textit{setColor}\ \textit{red}
\end{array}
$$

The function *storey* expects as first argument a function to draw the next floor and as second argument the width. For a demonstration see figure 2.4.

$$
\begin{array}{lll}
\textit{storey} & :: & (\textit{Numeric} \rightarrow \textit{Turtle}) \rightarrow \textit{Numeric} \rightarrow \textit{Turtle} \\
\textit{storey}\ r\ l & = & \textit{forward}\ l \mathbin{\&} \textit{toleft} \mathbin{\&} \textit{forward}\ l \mathbin{\&} \textit{toleft} \mathbin{\&} \textit{forward}\ l \\
 & & \&\textit{turn}\ 180 \mathbin{\&} r\ l \mathbin{\&} \textit{turn}\ (-45) \mathbin{\&} \textit{forward}\ (l * \textit{sqrt}\ 2) \\
 & & \&\textit{turn}\ (-135) \mathbin{\&} \textit{forward}\ l \mathbin{\&} \textit{turn}\ (-135) \\
 & & \&\textit{forward}\ (l * \textit{sqrt}\ 2) \mathbin{\&} \textit{turn}\ (-45)
\end{array}
$$

2.3 Trees

Trees are ubiquitous structures in computer sciences. *Functional* METAPOST implements them as special objects. The layout is automatically calculated from a formal specification of a tree.

A tree consists of nodes with arbitrary many edges which again end in nodes.

```
let canopy 0 l a _   = fork (turn a & forward l)
                            (turn (−a) & forward l)
    canopy n l a d   = fork (turn a & forward l
                             & canopy (n − 1) (l ∗ d) a d)
                            (turn (−a) & forward l
                             & canopy (n − 1) (l ∗ d) a d)
in rowSepBy 25 (map (setPen 0.2)
                    [canopy 8 30 80 0.67, canopy 8 30 100 0.62,
                     canopy 8 25 30 0.55, canopy 8 30 90 0.7])
```

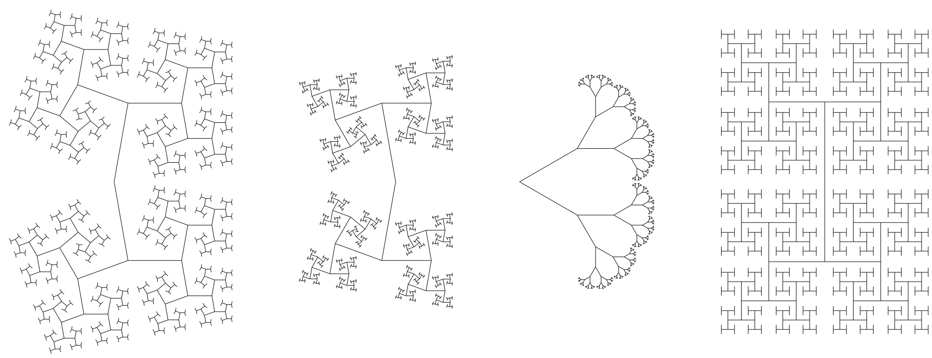


Figure 2.3: Fractal canopies, see [Man87], chapter 16.

```
    storey roof 20
  & home & penUp
  & forward 40 & fromPicture "Santa"
  & forward 20 & penDown
  & storey (storey (storey roof)) 15
    # setPen 2
```

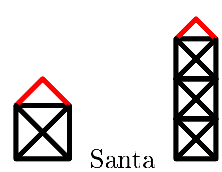
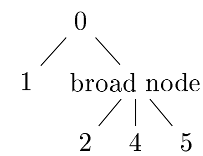


Figure 2.4: This picture consists of one turtle path. After drawing the first house the pen is raised, a picture with text added and the right house is drawn.

```
node    :: IsPicture a => a -> [Edge] -> Tree
edge    :: Tree -> Edge
```

This allows an easy description of a tree. The task of placement of nodes such that a "nice" picture arises is taken on by *functional* METAPOST.

```
tree1 = node "0" [edge (node "1" []),
                  edge (node "broad␣node" [edge (node "2" []),
                                           edge (node "4" []),
                                           edge (node "5" [])])]
```



The calculation of the layout follows the following set of rules:

① Nodes with the same distance to the root lie on a horizontal line.

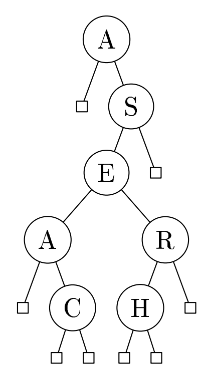
② A parent node is centered above its child nodes.

③ Equal subtrees have the same layout independent of their position in the tree.

At the same time the layout is adopted to the width and height of the pictures associated with the nodes. This is important to avoid overlaps and to achieve uniform distances.

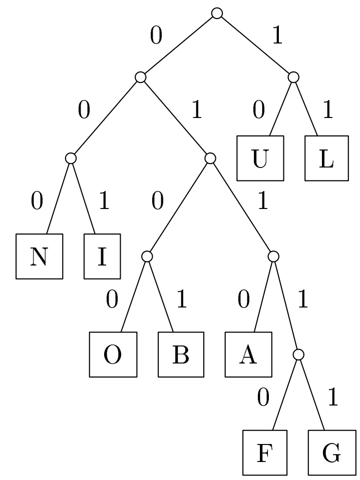
Pictures at the nodes can be framed, as in the following picture of a binary search tree. See [Sed92], page 242 for a worse layout of the same tree.

```
let enc s t = edge $ node (circle s) t
    nil = edge $ node (box empty) []
in node (circle "A")
        [nil,
         enc "S" [enc "E" [enc "A" [nil,
                                    enc "C" [nil, nil]],
                           enc "R" [enc "H" [nil, nil],
                                    nil]],
                  nil]]
```



Nodes have all attributes of pictures and edges have all attributes of paths. We can label edges in a HUFFMAN tree.

```
let inner l r = node (circle empty)
                     [edge l # setLabel 0.4 SE "0",
                      edge r # setLabel 0.4 SW "1"]
    leaf  s = node (box s) []
in inner (inner (inner (leaf "N")
                       (leaf "I"))
                (inner (inner (leaf "O")
                              (leaf "B"))
                       (inner (leaf "A")
                              (inner (leaf "F")
                                     (leaf "G")))))
         (inner (leaf "U")
                (leaf "L"))
```

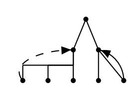


In addition we also allow crossing edges, generated by *cross*.

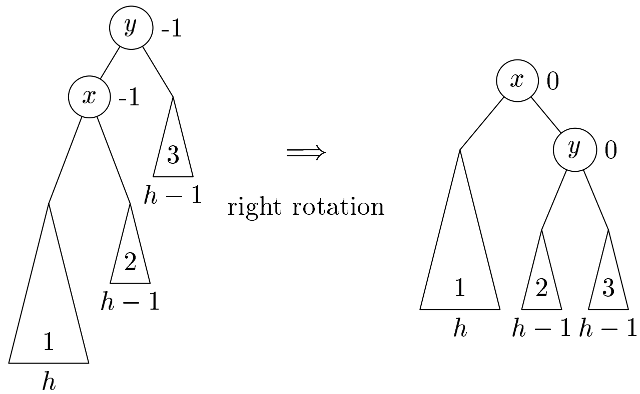
$$cross \quad :: \quad Point \rightarrow Edge$$

Crossing edges are simply added to the edge list of a node and run from this node to the given point. They do **not** influence the tree layout.

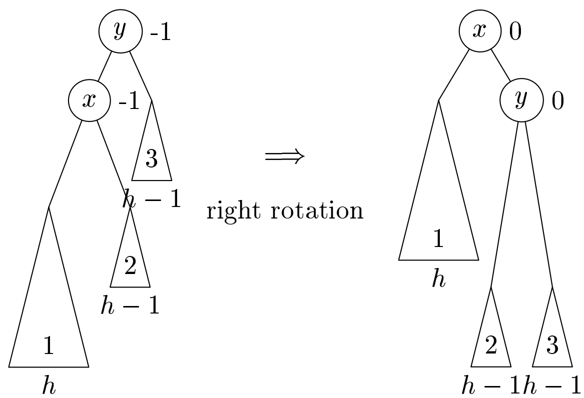
```
node2 [edge2 (node2
                [edge2s (node2
                           [upToRoot
                             # setPattern dashed
                             # setEndAngle 0
                             # setStartAngle 130]),
                 edge2s (node2 []),
                 edge2s (node2 [])]
                 # setAlign alignRight),
       edge2 (node2
                [edge2 (node2 []),
                 edge2 (node2 [upToRoot])]
                 # setAlign alignLeft)
                ]
where
node2 = node dot
edge2 = edge' (line (ref (This ◁ C)) (ref (Parent ◁ C)))
edge2s = edge' (stair (ref (This ◁ C)) (ref (Parent ◁ C)))
upToRoot = cross' (curve (ref This ◁ C)) (ref (Up 1 ◁ C))
                     # setStartAngle (90)
                     # setArrowHead default)
```



The automatic layout generation is quite efficient but sometimes a manual intervention is necessary as the next example shows.



Without manual intervention the layout would look wrong:



At the right tree the horizontal distance between the children of the root is too small. At the left tree the lower edges of the triangles are not aligned.

This can be corrected by changing vertical and horizontal distances between nodes:

$$
\begin{array}{lll}
\mathit{setDistH} & :: & \mathit{Distance} \to \mathit{Tree} \to \mathit{Tree} \\
\mathit{getDistH} & :: & \mathit{Tree} \to \mathit{Distance} \\
& & \\
\mathit{setDistV} & :: & \mathit{Distance} \to \mathit{Tree} \to \mathit{Tree} \\
\mathit{getDistV} & :: & \mathit{Tree} \to \mathit{Distance}
\end{array}
$$

There are two different possibilities to supply distances. The first defines the distance between the frames of the node pictures. This is the default if *setDistV* or *setDistH* are called simply with a number as parameter.

$$\mathit{distBorder} \quad :: \quad \mathit{Numeric} \to \mathit{Distance}$$

The second possibility is to supply the distance between the centers of the node pictures. This does not automatically avoid overlaps.

$$\mathit{distCenter} \quad :: \quad \mathit{Numeric} \to \mathit{Distance}$$

Now we can describe our example. We start with functions to create the triangular nodes with fixed heigth and width.

```
tri                    :: String → Picture
tri "1"                =  triangle "1" # setHeight 60 # setWidth 30
                            # label S (math "h")
tri s                  =  triangle s # setHeight 30 # setWidth 15
                            # label S (math "h-1")
```

The left tree needs a small modification. The distance between the two childs of the root node has to be enlarged. This is done by applying *setDistH* 16 to the root.

```
notRotated             :: Tree
notRotated             =  node (circle "$y$" # label E "-1")
                                [edge (node (circle "$x$" # label E "-1")
                                             [edge (node (tri "1") []),
                                              edge (node (tri "2") [])]),
                                 edge (node (tri "3") [])]
                            # setDistH 16
```

The difficulty for the second tree is to align the lower edges of the triangles. The attribute function *setDistV* (*distCenter* 30) achieves that the vertical distance of the triangles to the center of the y circle equals 30 points. This is what we want since the first triangle has a height of 60 points and the two others have half this height.

```
rightRotated           :: Tree
rightRotated           =  node (circle "$x$" # label E "0") [
                                edge (node (tri "1") []),
                                edge (node (circle "$y$" # label E "0")
                                             [edge (node (tri "2") []
                                                       # setDistV (distCenter 30)),
                                              edge (node (tri "3") []
                                                       # setDistV (distCenter 30))]
                                         # setDistH 10)]
                            # setDistH 20
```

Now we can describe the full picture:

```
rowSepBy 10 [toPicture notRotated,
             "$\\Longrightarrow$" ⊟ "right rotation",
             toPicture rightRotated]
```

When trees of some specific kind appear quite often one can introduce an abstraction for them. We show this for binary trees.

```
data BinTree           =  BNode BinTree Picture BinTree
                       |  BEmpty
```

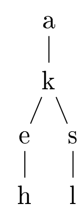
A converting function transforms a binary tree to the *Tree* type.

```
bin                    :: BinTree → Tree
bin BEmpty             =  node "empty bin" []
bin (BNode BEmpty p BEmpty)
```

```
                            = node p []
bin (BNode l p BEmpty)      = node p [edge (bin l)]
bin (BNode BEmpty p r)      = node p [edge (bin r)]
bin (BNode l p r)           = node p [edge (bin l), edge (bin r)]
```

The result of this attempt is disappointing. If a node has only one child we do not know whether this is a right or left child. The layout algorithm draws it exactly below the parent node.

```
let e = BEmpty
    n = BNode
in bin (n e
          (tex "a")
          (n (n e (tex "e") (n e (tex "h") e))
             (tex "k")
             (n (n e (tex "l") e) (tex "s") e)))
```



We add a node attribute to control the alignment of children.

```
setAlign                    :: AlignSons → Tree → Tree
getAlign                    :: Tree → AlignSons
```

For binary trees we have the two alignements

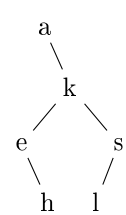
```
alignLeftSon, alignRightSon :: AlignSons
```

which let a single child node appear either to the right or to the left.

```
bin'                        :: BinTree → Tree
bin' BEmpty                 = node "empty␣bin" []
bin' (BNode BEmpty p BEmpty)
                            = node p []
bin' (BNode l p BEmpty)     = node p [edge (bin' l)]
                                # setAlign alignLeftSon
bin' (BNode BEmpty p r)     = node p [edge (bin' r)]
                                # setAlign alignRightSon
bin' (BNode l p r) = node p [edge (bin' l), edge (bin' r)]
```

With this modifications we get the binary tree as we wish it:

```
let e = BEmpty
    n = BNode
in bin' (n e
          (tex "a")
          (n (n e (tex "e") (n e (tex "h") e))
             (tex "k")
             (n (n e (tex "l") e) (tex "s") e)))
```

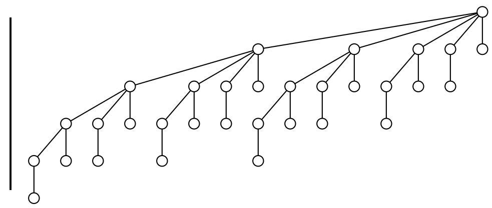


Another alignment is useful for binomial trees. The alignment types

```
alignLeft, alignRight       :: AlignSons
```

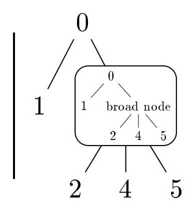
place the child nodes left or right adjusted below the parent node.

```
let ce = circle empty
    binom 0 = node ce []
    binom n = node ce [edge (binom i)
                      | i ← [(n − 1), (n − 2) . . 0]]
              # setAlign AlignRight
in binom 5
```



We want to emphasize that a node may be an arbitrary picture and that the layout of the tree is automatically accommodated to the size of the node. This even allows to draw trees where the nodes themselves are trees, as used for some special data structures (data–structural bootstrapping [BO96]).

```
node "0" [edge (node "1" []),
          edge (node (rbox 5 (scale 0.5 tree1)) [edge (node "2" []),
                                                  edge (node "4" []),
                                                  edge (node "5" [])])]
```



Till now the trees in the examples were rather small. Figure 2.5 shows something more complex, a tree from figure 44.2 in [Sed92] with 153 nodes. We refrain from showing the longish but straightforward description.

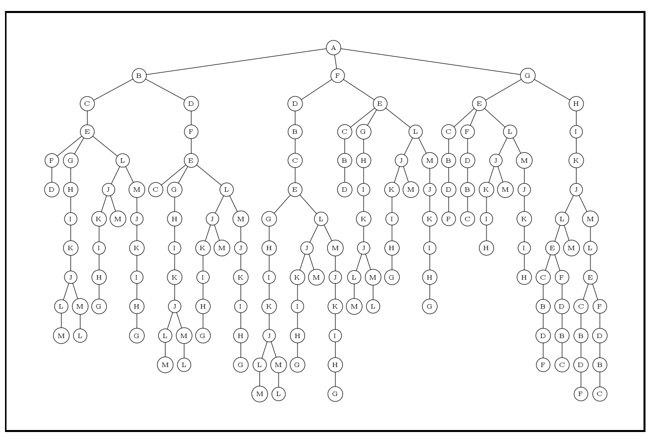


Figure 2.5: Depth first search for a HAMILTON cycle in a graph.

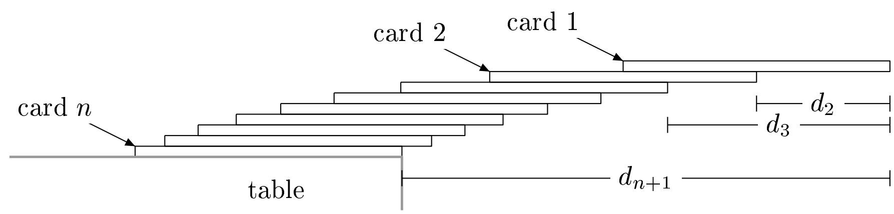
2.4 Further examples

Finally we want to demonstrate the abilities of *functional* METAPOST with some more complex examples.

Example 1

This example demonstrates the usefulness of the embedding into Haskell. The full power of Haskell is available for necessary calculations.

The book [GKP92] shows on page 259 a nice picture illustrating the problem to pile up n playing cards in such a way that the pile extends beyond the edge of the desk as far as possible.



The optimal distances d_2 to d_{n+1} are given by the series

$$d_{k+1} = \frac{(d_1+1)+(d_2+1)+\ldots+(d_k+1)}{k} \qquad , \text{for } 1 \leqslant k \leqslant n$$

This is equivalent to $\mathcal{H}_k = d_{k+1}$ where $\mathcal{H}_k$ denotes the harmonic series

$$\mathcal{H}_n = \sum_{k=1}^{n} \frac{1}{k} \qquad , \text{for } n \geqslant 0$$

. This can be calculated easily by Haskell:[3]

$$\begin{array}{lcl} \mathit{harmonic} & :: & [\mathit{Double}] \\ \mathit{harmonic} & = & 0 : [h + 1 \mathbin{/} k \mid (h, k) \leftarrow \mathit{zip}\ \mathit{harmonic}\ [1\,..]] \end{array}$$

As a next step we want to draw the card pile. The size of the cards should be easily changeable. We define some constants for it.

$$\begin{array}{lcl} \mathit{cardW}, \mathit{cardH} & :: & \mathit{Numeric} \\ \mathit{cardW} & = & 100 \\ \mathit{cardH} & = & 4 \end{array}$$

The horizontal coordinate of card number k is $\mathcal{H}_k$ multiplied by the card width.

$$\begin{array}{lcl} \mathit{cardX} & :: & \mathit{Numeric} \rightarrow \mathit{Numeric} \\ \mathit{cardX}\ k & = & -\mathit{cardW} * \mathit{fromDouble}\ (\mathit{harmonic}\ !!\ (\mathit{fromEnum}\ k)) \end{array}$$

The pile consist of nine cards, drawn as rectangles starting from the top.

[3] This definition has linear run time behaviour. A too naive approach leads to a quadratic run time:

$$\begin{array}{lcl} \mathit{harmonic}' & = & [h\ n \mid n \leftarrow [0\,..]] \\ \quad \textbf{where} & & \\ \quad h\ 0 & = & 0 \\ \quad h\ k & = & 1 \mathbin{/} k + h\ (k-1) \end{array}$$

$$
\begin{array}{lcl}
cards & :: & Canvas \\
cards & = & foldl\ (\&)\ relax \\
 & & \qquad [\,cdraw\ ((cardX\ \ n, cardH * (1-n)) \\
 & & \qquad\qquad\quad \text{--}\,(cardX\ \ n + cardW, cardH * (1-n)) \\
 & & \qquad\qquad\quad \text{--}\,(cardX\ \ n + cardW, -cardH * n) \\
 & & \qquad\qquad\quad \text{--}\,(cardX\ \ n, -cardH * n)\ \text{--}\ cycle) \\
 & & \qquad \mid n \leftarrow [1\,..\,9]]
\end{array}
$$

The table is a canvas graphics, too.

$$
\begin{array}{lcl}
table & :: & Canvas \\
table & = & cdraw\ ((-330, -9 * cardH) \\
 & & \qquad\quad \text{--}\,(cardW + cardX\ \ 9, -9 * cardH) \\
 & & \qquad\quad \text{--}\,(cardW + cardX\ \ 9, -14 * cardH) \\
 & & \qquad\quad \#\ setPen\ 1\ \#\ setColor\ 0.6) \\
 & \& & cdrop\ (-230, -12.0 * cardH)\ \texttt{"table"}
\end{array}
$$

The following function is generally useful for dimensioning. At the ends of the line are arrowheads with an opening angle of 180 degree. In the middle is a text label with white background.

$$
\begin{array}{lcl}
dimension & :: & (IsPath\ \ b, IsPath\ \ a) \Rightarrow a \rightarrow b \rightarrow String \rightarrow Path \\
dimension\ \ a\ \ b\ \ s & = & a\ \text{--}\ b \\
 & & \ \#\ setArrowHead\ \ marker \\
 & & \ \#\ setStartArrowHead\ \ marker \\
 & & \ \#\ setLabel\ 0.5\ \ C\ \ (math\ \ s\ \#\ setBGColor\ \ white) \\
\quad\textbf{where} & & \\
\quad marker & = & arrowHeadSize\ 3\ 180\ \#\ setArrowHeadStyle\ \ ahLine
\end{array}
$$

The necessary dimensioning uses again the $cardX$ function.

$$
\begin{array}{lcl}
dimensions & :: & Canvas \\
dimensions & = & cdraw\ (dimension\ (cardW + cardX\ \ 2, -4 * cardH) \\
 & & \qquad\qquad\qquad\quad (0, -4 * cardH)\ \texttt{"d\_2"}) \\
 & \& & cdraw\ (dimension\ (cardW + cardX\ \ 3, -6 * cardH) \\
 & & \qquad\qquad\qquad\quad (0, -6 * cardH)\ \texttt{"d\_3"}) \\
 & \& & cdraw\ (dimension\ (cardW + cardX\ \ 9, -11 * cardH) \\
 & & \qquad\qquad\qquad\quad (0, -11 * cardH)\ \texttt{"d\_\{n+1\}"})
\end{array}
$$

We also introduce a function for the frequent feature of a label where an arrow starts:

$$
\begin{array}{lcl}
description & :: & (IsPicture\ \ c, IsPath\ \ a, IsPath\ \ b) \\
 & \Rightarrow & a \rightarrow b \rightarrow c \rightarrow Path \\
description\ p_1\ p_2\ l & = & arrow\ p_1\ p_2 \\
 & & \ \#\ setLabel\ 0\ \ C\ \ (toPicture\ \ l\ \#\ setBGColor\ \ white)
\end{array}
$$

We need a special case:

$$
\begin{array}{lcl}
describeCard & :: & Point \rightarrow String \rightarrow Path \\
describeCard\ \ p\ \ l & = & description\ (p + vec\ (-30, 15))\ p\ (\texttt{"card␣"} +\!\!+\ l)
\end{array}
$$

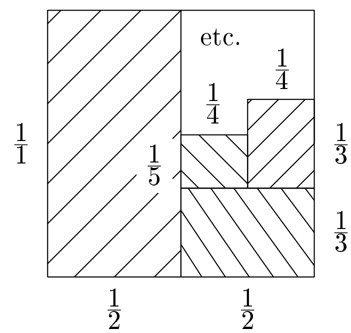
The complete picture consist of a canvas graphics including the card pile, the table, the dimensioning and the labels.

$$
\begin{aligned}
&\mathit{draw}\ [\mathit{describeCard}\ (\mathit{vec}\ (\mathit{cardX}\ 1, 0))\ \texttt{"\$1\$"},\\
&\qquad \mathit{describeCard}\ (\mathit{vec}\ (\mathit{cardX}\ 2, -\mathit{cardH}))\ \texttt{"\$2\$"},\\
&\qquad \mathit{describeCard}\ (\mathit{vec}\ (\mathit{cardX}\ 9, -8 * \mathit{cardH}))\ \texttt{"\$n\$"}]\\
&\quad (\mathit{cards}\ \&\ \mathit{table}\ \&\ \mathit{dimensions})
\end{aligned}
$$

Example 2

Page 66 of the book [GKP92] illustrated the problem to fill a unit square with rectangles of the sizes $\frac{1}{1} \times \frac{1}{2}$, $\frac{1}{2} \times \frac{1}{3}$, $\frac{1}{3} \times \frac{1}{4}$.

PostScript and therefore also METAPOST do not support hatchings or other filling patterns for areas. The book [Ado88] suggests to draw the fill pattern in a rectangle and to cut out the required area. This way is also possible in *functional* METAPOST.



We define a function *patternBox* which draws a rectangle with hatchings. The first two arguments are the coordinates of the lower left and the upper right corner. The next two arguments are functions which influence the direction of the hatching.

$$
\begin{aligned}
&\mathit{patternBox} && :: (\mathit{Numeric}, \mathit{Numeric}) \to (\mathit{Numeric}, \mathit{Numeric})\\
& && \to (\mathit{Numeric} \to (\mathit{Numeric}, \mathit{Numeric}))\\
& && \to (\mathit{Numeric} \to (\mathit{Numeric}, \mathit{Numeric})) \to \mathit{Canvas}\\
&\mathit{patternBox}\ (ax, ay)\ (bx, by)\ \mathit{fa}\ \mathit{fb} && \\
& && = \mathit{cdrop}\ (0.5 * (ax + bx), 0.5 * (ay + by))\\
& && \qquad (\mathit{cdraws}\ [\mathit{fa}\ i \mathbin{\text{--}} \mathit{fb}\ i \mid i \leftarrow [-50, -40 \mathinner{..} 200]]\\
& && \qquad \&\mathit{cclip}\ p\\
& && \qquad \&\mathit{cdraw}\ p)\\
&\qquad \textbf{where} && \\
&\qquad p && = \mathit{vec}\ (ax, ay) \mathbin{\text{--}} \mathit{vec}\ (bx, ay) \mathbin{\text{--}} \mathit{vec}\ (bx, by)\\
& && \quad \mathbin{\text{--}}\mathit{vec}\ (ax, by) \mathbin{\text{--}} \mathit{cycle}
\end{aligned}
$$

The fractions are printed in a slightly larger font. Their background is white so that the hatching avoids the fraction $\frac{1}{5}$.

$$
\begin{aligned}
&\mathit{frac} && :: \mathit{Show}\ a \Rightarrow (\mathit{Numeric}, \mathit{Numeric}) \to a \to \mathit{Canvas}\\
&\mathit{frac}\ p\ n && = \mathit{cdrop}\ p\ (\mathit{math}\ (\texttt{"\textbackslash\textbackslash frac\{\textbackslash\textbackslash textstyle␣1\}"}\\
& && \qquad\qquad + \!\!+ \texttt{"\{\textbackslash\textbackslash textstyle␣"} + \!\!+ \mathit{show}\ n + \!\!+ \texttt{"\}"})\\
& && \qquad \#\ \mathit{setBGColor}\ \mathit{white})
\end{aligned}
$$

```
    patternBox (0,0) (50,100)        (\n -> (0,200 - n * 1.5))  (\n -> (n * 1.5,200))
  & patternBox (50,0) (100,33.3)     (\n -> (0,n))              (\n -> (n,-50))
  & patternBox (75,33.3) (100,66.6)(\n -> (0,100 - n))          (\n -> (n,100))
  & patternBox (50,33.3) (75,53.3)  (\n -> (0,n))               (\n -> (n,0))
  & cdraw (vec (50,100) -- vec (100,100) -- vec (100,50))       -- upper right corner
  & frac (-10,50) 1
  & frac (25,-12) 2 & frac (75,-12) 2
  & frac (110,17) 3 & frac (110,50) 3
  & frac (62,65) 4 & frac (88,78) 4
  & frac (40,42) 5
  & cdrop (65,90) "etc."
```

Admittedly, the description of this picture is not very descriptive. It contains a lot of explicit coordinates. But sometimes this is the simplest way.

Example 3

Many data structures are realized by special trees. We will define special abstractions for 2–3–4 trees and red–black trees as described e.g. in [Oka98]. This also demonstrates how to modify the drawing of edges according to special needs.

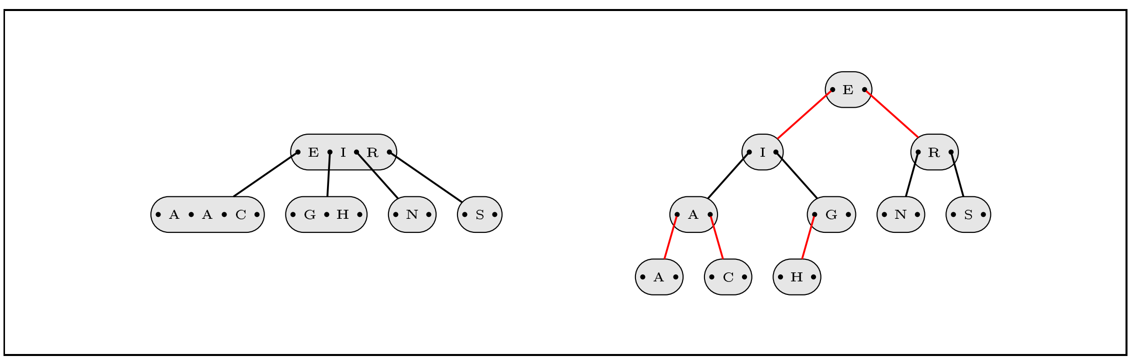


Figure 2.6: The same data represented as 2–3–4 tree and as red–black tree.

We start with the definition of 2–3–4 trees. A node may obtain none, two, three or four elements.

```
data Tree234 a  = Nil
                | Two (Tree234 a) a (Tree234 a)
                | Three (Tree234 a) a (Tree234 a) a (Tree234 a)
                | Four (Tree234 a) a (Tree234 a) a (Tree234 a) a (Tree234 a)
                  deriving (Show)
```

The trees from figure 2.6 can be described by this data structure:

```
rbtree  :: Tree234 String
rbtree  = Four (Four Nil "A" Nil "A" Nil "C" Nil)
               "E"
               (Three Nil "G" Nil "H" Nil)
```

$$
\begin{array}{l}
\texttt{"I"} \\
(\mathit{Two}\ \mathit{Nil}\ \texttt{"N"}\ \mathit{Nil}) \\
\texttt{"R"} \\
(\mathit{Two}\ \mathit{Nil}\ \texttt{"S"}\ \mathit{Nil})
\end{array}
$$

We will now write two functions which transform a tree of type *Tree234 String* to a tree of type *Tree*, either with 2–3–4 or with red–black layout.

Let's start with an utility to create small texts:

$$
\begin{array}{lcl}
\mathit{tiny} & :: & \mathit{String} \to \mathit{Picture} \\
\mathit{tiny}\ a & = & \mathit{tex}\ (\texttt{"\textbackslash\textbackslash tiny\textvisiblespace"} +\!\!+\ a)
\end{array}
$$

The particularity of these trees is that edges do not start at the center of the nodes. But this is no problem for us. We only need a possibility to name the starting points.

$$
\begin{array}{lcl}
\mathit{dotName} & :: & \mathit{String} \to \mathit{Frame} \\
\mathit{dotName}\ n & = & \mathit{dot}\ \#\ \mathit{setName}\ n
\end{array}
$$

The node pictures are rounded rectangles with some content:

$$
\begin{array}{lcl}
\mathit{tbox} & :: & \mathit{String} \to \mathit{Frame} \\
\mathit{tbox}\ s & = & \mathit{rbox}\ 8\ (\mathit{dotName}\ \texttt{"p1"} \boxminus\ \mathit{tiny}\ s \boxminus \mathit{dotName}\ \texttt{"p2"}) \\
\mathit{tbox2}\ s_1\ s_2 & = & \mathit{rbox}\ 8\ (\mathit{dotName}\ \texttt{"p1"} \boxminus \mathit{tiny}\ s_1 \boxminus \mathit{dotName}\ \texttt{"p2"} \\
 & & \qquad \boxminus \mathit{tiny}\ s_2 \boxminus \mathit{dotName}\ \texttt{"p3"}) \\
\mathit{tbox3}\ s_1\ s_2\ s_3 & = & \mathit{rbox}\ 8\ (\mathit{dotName}\ \texttt{"p1"} \boxminus \mathit{tiny}\ s_1 \boxminus \mathit{dotName}\ \texttt{"p2"} \\
 & & \qquad \boxminus \mathit{tiny}\ s_2 \boxminus \mathit{dotName}\ \texttt{"p3"} \boxminus \mathit{tiny}\ s_3 \\
 & & \qquad \boxminus \mathit{dotName}\ \texttt{"p4"})
\end{array}
$$

Now we define special edges which run from the current child to the center of the subpicture named $\texttt{'p'} : \mathit{show}\ n$ in the parent node. This shows how useful it is to be able to generate names out of different parts. The special names *This* and *Parent* are placeholders and change their reference according to the context for every edge. This allows a general description of the edges. More details can be found in section 3.16.3.

$$
\begin{array}{lcl}
\mathit{edgeN} & :: & \mathit{Int} \to \mathit{Tree} \to \mathit{Edge} \\
\mathit{edgeN}\ n & = & \mathit{edge}'\ (\mathit{ref}\ (\mathit{This} \triangleleft C)\ \text{--}\ \mathit{ref}\ (\mathit{Parent} \triangleleft \texttt{'p'} : \mathit{show}\ n \triangleleft C))
\end{array}
$$

It is important to choose the edge according to the number of the child node:

$$
\begin{array}{lcl}
\mathit{convert234} & :: & \mathit{Tree234}\ \mathit{String} \to \mathit{Tree} \\
\mathit{convert234}\ \mathit{Nil} & = & \mathit{node}\ (\mathit{box}\ \mathit{empty})\ [\,] \\
\mathit{convert234}\ (\mathit{Two}\ t_1\ a\ t_2) & = & \mathit{node}\ (\mathit{tbox}\ a)\ (\mathit{edge234}\ 1\ t_1\ (\mathit{edge234}\ 2\ t_2\ [\,])) \\
\mathit{convert234}\ (\mathit{Three}\ t_1\ a_1\ t_2\ a_2\ t_3) & & \\
 & = & \mathit{node}\ (\mathit{tbox2}\ a_1\ a_2)\ (\mathit{edge234}\ 1\ t_1 \\
 & & \qquad (\mathit{edge234}\ 2\ t_2 \\
 & & \qquad (\mathit{edge234}\ 3\ t_3\ [\,]))) \\
\mathit{convert234}\ (\mathit{Four}\ t_1\ a_1\ t_2\ a_2\ t_3\ a_3\ t_4) & & \\
 & = & \ldots \\
 & & \\
\mathit{edge234} & :: & \mathit{Int} \to \mathit{Tree234}\ \mathit{String} \to [\mathit{Edge}] \to [\mathit{Edge}] \\
\mathit{edge234}\ n\ \mathit{Nil}\ \mathit{cont} & = & \mathit{cont} \\
\mathit{edge234}\ n\ t\ \mathit{cont} & = & \mathit{edgeN}\ n\ (\mathit{convert234}\ t) : \mathit{cont}
\end{array}
$$

During conversion to the red–black representation a node may be mapped to several nodes of a binary tree. In the case of a 3–node we must ensure that the left child is aligned to the left.

$$
\begin{array}{lcl}
\mathit{convertRB} & :: & \mathit{Tree234}\ \mathit{String} \rightarrow \mathit{Tree} \\
\mathit{convertRB}\ \mathit{Nil} & = & \mathit{node}\ (\mathit{box}\ \mathit{empty})\ [\,] \\
\mathit{convertRB}\ (\mathit{Two}\ t_1\ a\ t_2) & = & \mathit{node}\ (\mathit{tbox}\ a)\ (\mathit{edgeRB}\ 1\ t_1\ (\mathit{edgeRB}\ 2\ t_2\ [\,])) \\
\mathit{convertRB}\ (\mathit{Three}\ t_1\ a_1\ t_2\ a_2\ t_3) & & \\
 & = & \mathit{node}\ (\mathit{tbox}\ a_1)\ (\mathit{map}\ (\mathit{setColor}\ \mathit{red}) \\
 & & \qquad\qquad\qquad (\mathit{edgeRB}\ 1\ (\mathit{Two}\ t_1\ a_2\ t_2) \\
 & & \qquad\qquad (\mathit{edgeRB}\ 2\ t_3\ [\,])))\ \#\ \mathit{setAlign}\ \mathit{alignLeftSon} \\
\mathit{convertRB}\ (\mathit{Four}\ t_1\ a_1\ t_2\ a_2\ t_3\ a_3\ t_4) & & \\
 & = & \mathit{node}\ (\mathit{tbox}\ a_1) \\
 & & \quad (\mathit{map}\ (\mathit{setColor}\ \mathit{red})\ (\mathit{edgeRB}\ 1\ (\mathit{Two}\ t_1\ a_2\ t_2) \\
 & & \qquad\qquad\qquad (\mathit{edgeRB}\ 2\ (\mathit{Two}\ t_3\ a_3\ t_4)\ [\,])))
\end{array}
$$

Now we add a thicker pen for the edges and a gray background shadow for the node pictures. This is done by the functions *forEachPic* and *forEachEdge* which apply a function to all nodes or edges of a tree.

$$
\begin{array}{ll}
 & \mathit{forEachPic}\ (\mathit{setBGColor}\ 0.9)\ (\mathit{forEachEdge}\ (\mathit{setPen}\ 0.75)\ (\mathit{convert234}\ \mathit{rbtree})) \\
\Box\!\Box & \mathit{hspace}\ 50 \\
\Box\!\Box & \mathit{forEachPic}\ (\mathit{setBGColor}\ 0.9)\ (\mathit{forEachEdge}\ (\mathit{setPen}\ 0.75)\ (\mathit{convertRB}\ \mathit{rbtree}))
\end{array}
$$

Example 4

This example will demonstrate once more the usefulness and ability of systems of equations. Our task is to draw a bracket between two given points and to add a label. The bracket will be drawn by a calligraphic pen, such that the ends are sharp and the straight parts are thicker. Therefore the pen must be rotated. Another difficulty is the placement of the label. The cusp of the bracket should point to the center of the label. Figure 2.7 shows how the position of the label depends on the angle in which the bracket is drawn and how the linewidth of the bracket depends on its size.

We start with an expression which describes the path of the bracket. Using *define* we use equations to describe the positions of five points. They fix the bracket path. The variable *var* `"angle"` is the angle defined by the points *pl* and *pr*. The width *var* `"d"` of the pen is 5 bp. For small brackets, when the distance between *pl* and *pr* is smaller than 20 bp, the width is set to 5/4 bp.

The positioning of the label has to depend on the angle *var* `"angle"`. This condition can not be formulated as part of the path. We place a modified label at the reference point C and translate C to some point at the boundary depending on the angle.

For this we use the *overlay'* function which allows to choose the bounding box of the combined picture. We combine the empty picture with the label and choose – by the argument (*Just* 0) – the bounding box of the first picture, i.e. the one of *empty*. The function *label* creates a picture, the reference point C of which (depending on the *var* `"angle"`) is placed at one of the reference points N, NE, ... of the picture l.

In this way we get a function for a path in form of a bracket with some "intelligence". The user can employ it like any other path.

```
let bracket :: IsPicture a ⇒ a → (Point, Point) → Path
    bracket l (pl, pr) = define [var "angle" ≐ angle (pl − pr),
                                 var "d" ≐ cond (dist pl pr <̇ 20) (dist pl pr / 4) 5,
                                 ref "vecl" ≐ var "d" ∗̇ dir (var "angle" − 135),
                                 ref "vecr" ≐ var "d" ∗̇ dir (var "angle" − 45),
                                 ref "mid" ≐ med 0.5 pl pr
                                         + (1.41 ∗ var "d") ∗̇ dir (var "angle" − 90),
                                 ref "midl" ≐ ref "mid" − ref "vecl",
                                 ref "midr" ≐ ref "mid" − ref "vecr"]
                                (pl ... pl + ref "vecl" --- ref "midl" ... ref "mid"
                                 & ref "mid" ... ref "midr" --- pr + ref "vecr" ... pr
                                # setPen (penCircle (0.001, var "d" / 5) (var "angle"))
                                # setLabel 0.5 C label)
      where
      label = overlay' [var "angle" ≐ angle (pl − pr),
                        ref (0 ◃ C) ≐ cond (var "angle" <̇ −175.5
                                      + 175.5 <̇ var "angle") (ref (1 ◃ S))
                                    (cond (var "angle" <̇ −112.5) (ref (1 ◃ SE))
                                     (cond (var "angle" <̇ −67.5) (ref (1 ◃ E))
                                      (cond (var "angle" <̇ −22.5) (ref (1 ◃ NE))
                                       (cond (var "angle" <̇ 22.5) (ref (1 ◃ N))
                                        (cond (var "angle" <̇ 67.5) (ref (1 ◃ NW))
                                         (cond (var "angle" <̇ 112.5) (ref (1 ◃ W))
                                                                      (ref (1 ◃ SW))
                                    )))))))] (Just 0) [empty, toPicture l]
in [bracket a (5 ∗ dir a, 80 ∗ dir a) | a ← [0, 45 .. 315]]
  □□ [bracket y (vec (x, 0), vec (x, y)) | (x, y) ← zip [0, 30 ..] (5 : 10 : 15 : [20, 30 .. 60])]
```

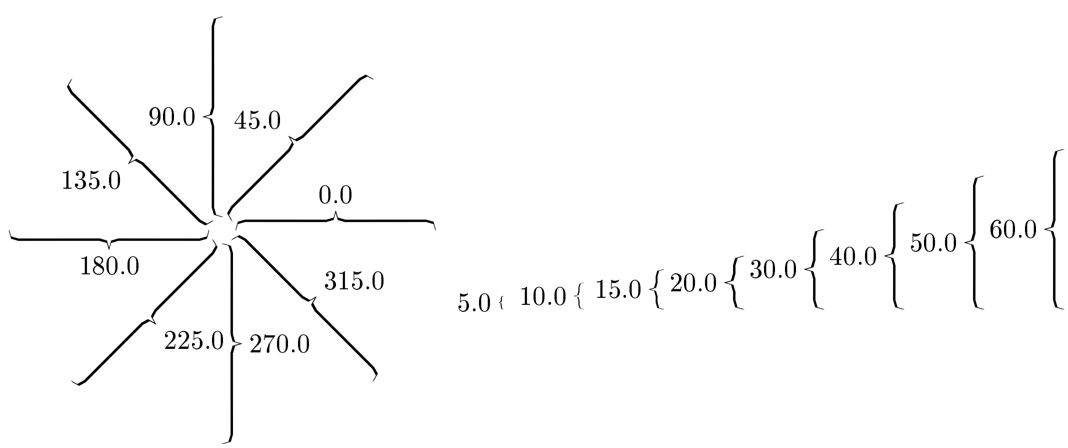


Figure 2.7: A path in form of a bracket is generated between the points pl and pr. The picture l is added as label.

Chapter 3

More *functional* METAPOST commands

This chapter lists the remaining, not yet mentioned *functional* METAPOST features.

Each part starts with the relevant classes and instances. Then we list the most important exported types and functions. The chapter is written in a mostly reference-like style.

The order of sections follows the one from the first chapter.

3.1 Atomic Pictures

Picture is instance of: *IsPicture*, *HasColor*, *HasBGColor*, *HasName* and *HasDefine*.

Instances of *IsPicture* are: *Picture*, *Char*, *Int*, *Integer*, *Numeric*, $\mathit{IsPicture}\ a \Rightarrow \mathit{IsPicture}\ [a]$, $()$, $(\mathit{IsPicture}\ a, \mathit{IsPicture}\ b) \Rightarrow \mathit{IsPicture}\ (a, b)$, $(\mathit{IsPicture}\ a, \mathit{IsPicture}\ b, \mathit{IsPicture}\ c) \Rightarrow \mathit{IsPicture}\ (a, b, c)$, *Path*, *Area*, *Frame*, *Tree*, *Canvas*, *Turtle*, ...

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Picture} & = & \mathit{Attributes}\ \mathit{Attrib}\ \mathit{Picture} \\
 & \mid & \mathit{Overlay}\ [\mathit{Equation}]\ (\mathit{Maybe}\ \mathit{Int})\ [\mathit{Picture}] \\
 & \mid & \mathit{Define}\ [\mathit{Equation}]\ \mathit{Picture} \\
 & \mid & \mathit{Frame}\ \mathit{FrameAttrib}\ [\mathit{Equation}]\ \mathit{Path}\ \mathit{Picture} \\
 & \mid & \mathit{Draw}\ [\mathit{Path}]\ \mathit{Picture} \\
 & \mid & \mathit{Fill}\ [\mathit{Area}]\ \mathit{Picture} \\
 & \mid & \mathit{Clip}\ \mathit{Path}\ \mathit{Picture} \\
 & \mid & \mathit{Empty}\ \mathit{Numeric}\ \mathit{Numeric} \\
 & \mid & \mathit{Tex}\ \mathit{String} \\
 & \mid & \mathit{Text}\ \mathit{String} \\
 & \mid & \mathit{BitLine}\ \mathit{Point}\ \mathit{BitDepth}\ \mathit{String} \\
 & \mid & \mathit{PTransform}\ \mathit{Transformation}\ \mathit{Picture} \\
 & \mid & \mathit{TrueBox}\ \mathit{Picture} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read}) \\
\textbf{class}\ \mathit{HasPicture}\ a\ \textbf{where} & & \\
\quad \mathit{fromPicture} & :: & (\mathit{IsPicture}\ b) \Rightarrow b \to a
\end{array}
$$

There are even more constants of length: Didôt–point, big point (PostScript point), Pica, Cicero, and Inch.

$$
\begin{array}{lcl}
dd, bp, pc, cc, inch & :: & Numeric \\
dd & = & 1.06601 \\
bp & = & 1 \\
pc & = & 11.95517 \\
cc & = & 12.79213 \\
inch & = & 72
\end{array}
$$

3.2 Frames

Frame is instance of: *IsPicture*, *HasColor*, *HasBGColor*, *HasPen*, *HasPattern*, *HasShadow*, *HasDXY*, *HasExtent*, *HasName* and *IsHideable*.

$$
\begin{array}{lcl}
\textbf{data}\ Frame & = & Frame'\ FrameAttrib\ ExtentAttrib\ Picture \\
 & & \textbf{deriving}\ Show \\
\textbf{class}\ HasDXY\ a\ \textbf{where} & & \\
\quad setDX & :: & Numeric \rightarrow a \rightarrow a \\
\quad getDX & :: & a \rightarrow Maybe\ Numeric \\
\quad setDY & :: & Numeric \rightarrow a \rightarrow a \\
\quad getDY & :: & a \rightarrow Maybe\ Numeric \\
\textbf{class}\ HasExtent\ a\ \textbf{where} & & \\
\quad setWidth & :: & Numeric \rightarrow a \rightarrow a \\
\quad removeWidth & :: & a \rightarrow a \\
\quad getWidth & :: & a \rightarrow Maybe\ Numeric \\
\quad setHeight & :: & Numeric \rightarrow a \rightarrow a \\
\quad removeHeight & :: & a \rightarrow a \\
\quad getHeight & :: & a \rightarrow Maybe\ Numeric \\
\textbf{class}\ HasShadow\ a\ \textbf{where} & & \\
\quad setShadow & :: & (Numeric, Numeric) \rightarrow a \rightarrow a \\
\quad clearShadow & :: & a \rightarrow a \\
\quad getShadow & :: & a \rightarrow Maybe\ (Numeric, Numeric) \\
\textbf{class}\ HasExtent\ a\ \textbf{where} & & \\
\quad setWidth & :: & Numeric \rightarrow a \rightarrow a \\
\quad removeWidth & :: & a \rightarrow a \\
\quad getWidth & :: & a \rightarrow Maybe\ Numeric \\
\quad setHeight & :: & Numeric \rightarrow a \rightarrow a \\
\quad removeHeight & :: & a \rightarrow a \\
\quad getHeight & :: & a \rightarrow Maybe\ Numeric \\
\textbf{class}\ IsHideable\ a\ \textbf{where} & & \\
\quad hide & :: & a \rightarrow a
\end{array}
$$

In addition to the already mentioned frames we have frames for triangles (with angle on top as parameter) and quadratic frames rotated by 45 degree:

$triAngle :: IsPicture\ a \Rightarrow Numeric \rightarrow a \rightarrow Frame$
$diamond :: IsPicture\ a \Rightarrow a \rightarrow Frame$

and fuzzy frames. Here the first two parameters are starting values for a random number generator.

$fuzzy :: IsPicture\ a \Rightarrow Int \rightarrow Int \rightarrow a \rightarrow Frame$

People who draw graphs of data bases like little barrels:

$drum :: IsPicture\ a \Rightarrow a \rightarrow Frame$

It is possible to use shadows and clipping on all those frames. Figure 3.1 shows the four additional frame types:

```
rowSepBy cm [triAngle 90 (tex "\\texttt{triAngle␣90}") # setDX 16,
             diamond (tex "\\texttt{diamond}"),
             fuzzy 1 2 (tex "\\texttt{fuzzy␣1␣2}"),
             drum (tex "\\texttt{drum}")]
```

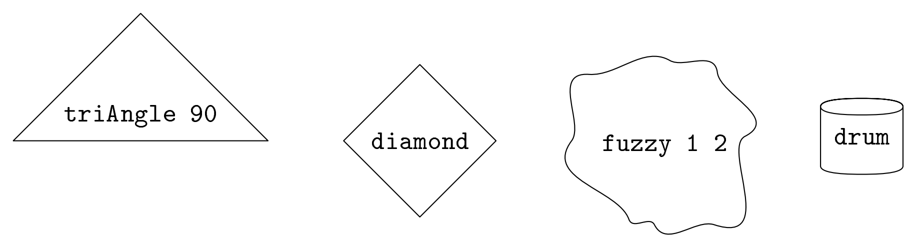


Figure 3.1: Four additional frame types.

3.3 Combination of pictures

The *matrix* function is a special case of the more general *matrixAlignSepBy*, which allows to align the items of the matrix separately, as in figure 3.2.

data *Cell* = *Cell Dir Picture*
  deriving *Show*

data *Cell* = *Cell Dir Picture*
  deriving *Show*

$$\begin{array}{lcl} cell' & :: & IsPicture\ a \Rightarrow Dir \rightarrow a \rightarrow Cell \\ matrixSepBy & :: & IsPicture\ a \Rightarrow Numeric \rightarrow Numeric \rightarrow [[a]] \rightarrow Picture \\ matrixAlign & :: & [[Cell]] \rightarrow Picture \end{array}$$

```
matrixAlignSepBy 10 10 [[cell' C (math "\\setminus"), cell' W "left␣adjusted",
                          cell' W "centered", cell' W "right␣adjusted"],
                         [cell' W ("vertical" ⊟ "top"),
                          cell' NW "NW", cell' N "N", cell' NE "NE"],
                         [cell' W ("vertikal" ⊟ "centered"),
                          cell' W "W", cell' C "C", cell' E "E"],
                         [cell' W ("vertikal" ⊟ "bottom"),
                          cell' SW "SW", cell' S "S", cell' SE "SE"]]
```

\	left adjusted	centered	right adjusted
vertical top	NW	N	NE
vertikal centered	W	C	E
vertikal bottom	SW	S	SE

Figure 3.2: The items of a matrix can be aligned separately. For clarity the space between rows and columns is shown in gray.

matrixAlignSepBy	::	$Numeric \to Numeric \to [[Cell]] \to Picture$
rowAlign	::	$[Cell] \to Picture$
columnAlign	::	$[Cell] \to Picture$
rowAlignSepBy	::	$Numeric \to [Cell] \to Picture$
columnAlignSepBy	::	$Numeric \to [Cell] \to Picture$

at	::	$(IsPicture\ a, IsPicture\ b)$
label	::	$(IsPicture\ a, IsPicture\ b) \Rightarrow Dir \to a \to b \to Picture$
ooalign	::	$IsPicture\ a \Rightarrow [a] \to Picture$ -- siehe LaTeX

3.4 Paths

Instances of *IsPath*: *Path*, *Point*, *Name*, $IsPath\ a \Rightarrow IsPath\ [a]$, *Char*, $(Num\ a, Num\ b, Real\ a, Real\ b) \Rightarrow IsPath\ (a, b)$.

Path is instance of the classes: *IsPicture*, *IsPath*, *HasLabel*, *HasConcat*, *HasColor*, *HasPattern*, *HasPen*, *IsHideable*, *HasArrowHead*, *HasStartEndDir*, *HasJoin*, *HasStartEndCut* and *HasDefine*.

$$
\begin{array}{lll}
\textbf{class } \mathit{HasLabel}\ a\ \textbf{where} & & \\
\quad \mathit{setLabel} & :: & \mathit{IsPicture}\ b \Rightarrow \mathit{Double} \to \mathit{Dir} \to b \to a \to a \\
\quad \mathit{removeLabel} & :: & a \to a \\
\textbf{class } \mathit{HasConcat}\ a\ \textbf{where} & & \\
\quad (\&) & :: & a \to a \to a
\end{array}
$$

Individual path segments can be made invisible using the *hide* function.

$$
\begin{array}{lll}
\textbf{class } \mathit{IsHideable}\ a\ \textbf{where} & & \\
\quad \mathit{hide} & :: & a \to a
\end{array}
$$

The base points of paths have many attributes which control the drawing of path segments. A value of *curl* larger than one gives a stronger curvature as normal, a value smaller than one a weaker curvature. Figure 3.8 shows an application of *setEndCurl*.

$$
\begin{array}{lll}
\textbf{class } \mathit{HasStartEndDir}\ a\ \textbf{where} & & \\
\quad \mathit{setStartAngle} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{setEndAngle} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{setStartCurl} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{setEndCurl} & :: & \mathit{Numeric} \to a \to a \\
\quad \mathit{setStartVector} & :: & \mathit{Point} \to a \to a \\
\quad \mathit{setEndVector} & :: & \mathit{Point} \to a \to a \\
\quad \mathit{removeStartDir} & :: & a \to a \\
\quad \mathit{removeEndDir} & :: & a \to a \\
\textbf{class } \mathit{HasJoin}\ a\ \textbf{where} & & \\
\quad \mathit{setJoin} & :: & \mathit{BasicJoin} \to a \to a \\
\quad \mathit{getJoin} & :: & a \to \mathit{BasicJoin}
\end{array}
$$

Path segments can be clipped by arbitrary bounding boxes of pictures at their start or end. For path constructors connecting reference points the functions *setStartCut* or *setEndStartCut*, resp. are called automatically to clip the path at the bounding boxes of the pictures the reference points belong to. If you want the path segment to enter the picture, use the functions *removeStartCut* or *removeEndCut*.

For example, the start of the path segment $\mathit{ref}\ (n_1 \triangleleft C)$ -- $(\mathit{ref}\ (n_2 \triangleleft C) + \mathit{vec}\ (4,0))$ is cut at the bounding box of the picture n_1, but the end is not cut at n_2. This could be achieved by applying $\mathit{setEndCut}\ n_2$ to the path segment.

In order to modify figure 2.6 such that the edges are not drawn up to the points, we simply redefine the edge function:

$$
\begin{array}{lll}
\mathit{edgeN} & :: & \mathit{Int} \to \mathit{Tree} \to \mathit{Edge} \\
\mathit{edgeN}\ n & = & \mathit{edge}'\ (\mathit{ref}\ (\mathit{This} \triangleleft C)\ \text{--}\ \mathit{ref}\ (\mathit{Parent} \triangleleft \texttt{'p'}\ : \mathit{show}\ n \triangleleft C))
\end{array}
$$

Figure 3.3 shows the result.

$$
\begin{array}{lll}
\textbf{class } \mathit{HasStartEndCut}\ a\ \textbf{where} & & \\
\quad \mathit{setStartCut} & :: & \mathit{IsName}\ b \Rightarrow b \to a \to a \\
\quad \mathit{removeStartCut} & :: & a \to a \\
\quad \mathit{setEndCut} & :: & \mathit{IsName}\ b \Rightarrow b \to a \to a \\
\quad \mathit{removeEndCut} & :: & a \to a
\end{array}
$$

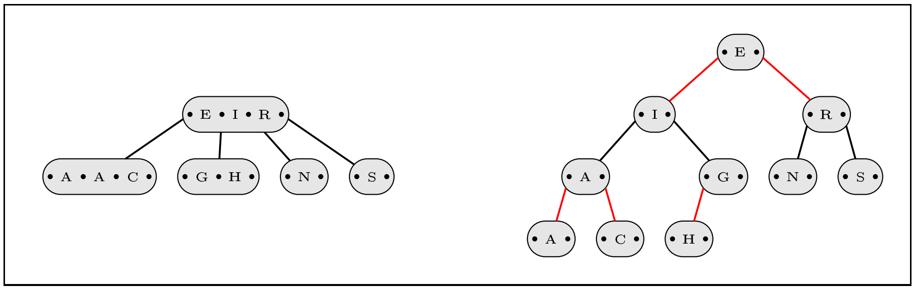


Figure 3.3: A modification of figure 2.6.

```
data Path                  = PathBuildCycle Path Path
                           | PathTransform Transformation Path
                           | PathPoint Point
                           | PathCycle
                           | PathJoin Path PathElemDescr Path
                           | PathEndDir Point Dir'
                           | PathDefine [Equation] Path
                             deriving (Eq, Show, Read)
data Dir'                  = DirEmpty
                           | DirCurl Numeric
                           | DirDir Numeric
                           | DirVector Point
                             deriving (Eq, Show, Read)

data PathElemDescr         = PathElemDescr{
                                   peColor :: Color,
                                   pePen :: Pen,
                                   peArrowHead :: Maybe ArrowHead,
                                   peSArrowHead :: Maybe ArrowHead,
                                   pePattern :: Pattern,
                                   peVisible :: Bool,
                                   peStartCut,
                                   peEndCut :: Maybe CutPic,
                                   peStartDir,
                                   peEndDir :: Dir',
                                   peJoin :: BasicJoin,
                                   peLabels :: [PathLabel]}
                             deriving (Eq, Read)

joinCat, joinFree, joinBounded,
joinStraight, joinTense     :: BasicJoin
joinTension                 :: Tension → BasicJoin
joinTensions                :: Tension → Tension → BasicJoin
joinControl                 :: Point → BasicJoin
joinControls                :: Point → Point → BasicJoin
```

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{BasicJoin} & = & \mathit{BJCat} \\
& \mid & \mathit{BJFree} \\
& \mid & \mathit{BJBounded} \\
& \mid & \mathit{BJStraight} \\
& \mid & \mathit{BJTense} \\
& \mid & \mathit{BJTension}\ \mathit{Tension} \\
& \mid & \mathit{BJTension2}\ \mathit{Tension}\ \mathit{Tension} \\
& \mid & \mathit{BJControls}\ \mathit{Point} \\
& \mid & \mathit{BJControls2}\ \mathit{Point}\ \mathit{Point} \\
& & \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{tension}, \mathit{tensionAtLeast} & :: & \mathit{Numeric} \to \mathit{Tension}
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Tension} & = & \mathit{Tension}\ \mathit{Numeric} \\
& \mid & \mathit{TensionAtLeast}\ \mathit{Numeric} \\
& & \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

Two twice intersecting paths can be used to build a cyclic path between the intersection points, as demonstrated in figure 3.4.

$$
\begin{array}{lcl}
\mathit{buildCycle} & :: & (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \to b \to \mathit{Path}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{pathLength} & :: & \mathit{Num}\ a \Rightarrow \mathit{Path} \to a \\
\mathit{forEachPath} & :: & (\mathit{PathElemDescr} \to \mathit{PathElemDescr}) \to \mathit{Path} \to \mathit{Path} \\
\mathit{line} & :: & (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \to b \to \mathit{Path} \\
\mathit{curve} & :: & (\mathit{IsPath}\ a, \mathit{IsPath}\ b) \Rightarrow a \to b \to \mathit{Path} \\
\mathit{arrow} & :: & (\mathit{IsPath}\ b, \mathit{IsPath}\ a) \Rightarrow a \to b \to \mathit{Path}
\end{array}
$$

Transformations can be applied not only to pictures but to paths, too. See figure 3.4.

$$
\begin{array}{lcl}
\mathit{transformPath} & :: & \mathit{Transformation} \to \mathit{Path} \to \mathit{Path} \\
\mathit{fullcircle}, \mathit{halfcircle}, & & \\
\mathit{quartercircle}, \mathit{unitsquare} & :: & \mathit{Path}
\end{array}
$$

3.5 Names

Instances of *IsName*: *Name*, *Int*, *Char*, *Dir* and $\mathit{IsName}\ a \Rightarrow \mathit{IsName}\ [a]$.

Instances of *HasName*: *Picture*, *Frame* and *Tree*.

$$
\begin{array}{lcl}
\mathbf{class}\ \mathit{HasName}\ a\ \mathbf{where} & & \\
\quad \mathit{setName} & :: & \mathit{IsName}\ b \Rightarrow b \to a \to a \\
\quad \mathit{getNames} & :: & a \to [\mathit{Name}]
\end{array}
$$

The function *enumPics* enumerates a list of n pictures. The first picture gets the name 0, the last the name $n-1$.

$$
\begin{array}{lcl}
\mathit{enumPics} & :: & \mathit{HasName}\ a \Rightarrow [a] \to [a]
\end{array}
$$

Figure 3.5 gives an example of referencing variables by a hierarchy of names.

```
bsp5          = box (math "U" □□ ooalign [toPicture [cArea a 0.7,
                                                    cArea b 0.7,
                                                    cArea ab 0.4],
                                          bOverA])
      where
      cArea a c = toArea a # setColor c
      bOverA    = column [math "B" # setBGColor white,
                          vspace 50,
                          math "A" # setBGColor white]
      a         = transformPath (scaled 30) fullcircle
      b         = transformPath (scaled 30 & shifted (0, −30))
                          fullcircle
      ab        = buildCycle a b
```

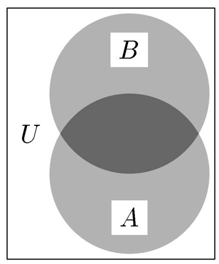


Figure 3.4: The function *buildCycle* creates a cycle from two intersecting paths. Compare Fig. 22 in [Hob92].

3.6 Numbers and points

```
data Point        = PointPic' Int Dir
                  | PointVar' Int Int
                  | PointVarArray' Int Int
                  | PointTrans' Point [Int]
                  | PointVar Name
                  | PointVec (Numeric, Numeric)
                  | PointMediate Numeric Point Point
                  | PointDirection Numeric
                  | PointWhatever
                  | PointPPP FunPPP Point Point
                  | PointNMul Numeric Point
                  | PointNeg Point
                  | PointCond Boolean Point Point
                    deriving (Eq, Show, Read, Ord)
```

```
let pointerChain dx ps = draw (backarrow : chainarrows)
                              (rowSepBy dx [b # setName (i :: Int)
                                           | (b, i) ← zip (map recBox ps) [0 ..]])
    where
    n = length ps
    backarrow = arrow (ref (n − 1 ◁ "bullet" ◁ C))
                      (ref (n − 1 ◁ "bullet" ◁ C) + vec (0, 20))
                       --- arrow (ref (0 ◁ W) + vec (0, 20)) (ref (0 ◁ W))
    chainarrows = [arrow (ref (i ◁ "bullet" ◁ C)) (ref (i + 1 ◁ W))
                  | i ← [0 .. n − 2]]
    recBox a = (box a # setHeight 16)
               ▭▭ (box (bullet # setName "bullet") # setHeight 16 # setWidth 16)
in pointerChain 25 ["42", "2", "3", "1109"]
```

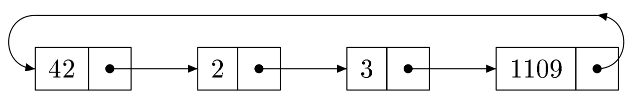


Figure 3.5: A linked list

```
data Numeric          = NumericVar′ Int Int
                      | NumericArray′ Int Int
                      | NumericVar Name
                      | Numeric Double
                      | NumericWhatever
                      | NumericDist Point Point
                      | NumericMediate Numeric Numeric Numeric
                      | NumericPN FunPN Point
                      | NumericNN FunNN Numeric
                      | NumericNNN FunNNN Numeric Numeric
                      | NumericNsN FunNsN [Numeric]
                      | NumericCond Boolean Numeric Numeric
                        deriving (Eq, Show, Read, Ord)

class HasCond a where
    cond              :: Boolean → a → a → a


(∗̇)                   :: Numeric → Point → Point

boolean               :: Bool → Boolean
```

The constants *up*, *right*, *down* and *left* denote the corresponding unit vectors.

$$
\begin{array}{lcl}
\mathit{up}, \mathit{down}, \mathit{left}, \mathit{right} & :: & \mathit{Point} \\
\mathit{up} & = & \mathit{vec}\ (0, 1) \\
\mathit{down} & = & \mathit{vec}\ (0, -1) \\
\mathit{left} & = & \mathit{vec}\ (-1, 0) \\
\mathit{right} & = & \mathit{vec}\ (1, 0)
\end{array}
$$

3.7 Symbolic equations

Instances of *HasDefine*: *Picture*, *Path* and *Area*.

$$
\begin{array}{lcl}
\mathit{equations} & :: & [\mathit{Equation}] \to \mathit{Equation} \\
\mathit{width}, \mathit{height} & :: & \mathit{IsName}\ a \Rightarrow a \to \mathit{Numeric}
\end{array}
$$

A variable in another system of equations can be referenced by prefixing the variable name with the function *global*.

$$
\begin{array}{lcl}
\mathit{global} & :: & \mathit{IsName}\ a \Rightarrow a \to \mathit{Name}
\end{array}
$$

The *overlay* function generates a bounding box enclosing all pictures. This is not always what we want. When one adds a label to a picture it may be useful not to increase the bounding box by the label, such that a second label is not placed with some additional distance. This is possible using *overlay'* which has an additional argument. For the value *Nothing* of this argument we get the functionality of *overlay*. For the value *Just b* the resulting bounding box is the one of picture no. $b + 1$ of the parameter list.

$$
\begin{array}{lcl}
\mathit{overlay}' & :: & \mathit{IsPicture}\ a \Rightarrow [\mathit{Equation}] \to \mathit{Maybe}\ \mathit{Int} \to [a] \\
\mathit{overlay} & :: & \mathit{IsPicture}\ a \Rightarrow [\mathit{Equation}] \to [a] \to \mathit{Picture} \\
\mathit{overlay}\ \mathit{eqs}\ \mathit{ps} & = & \mathit{overlay}'\ \mathit{eqs}\ \mathit{Nothing}\ \mathit{ps}
\end{array}
$$

3.8 Colors

Color is Instance of: *Num* and *Fractional*.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Color} & = & \mathit{DefaultColor} \\
 & | & \mathit{Color}\ \mathit{Double}\ \mathit{Double}\ \mathit{Double} \\
 & | & \mathit{Graduate}\ \mathit{Color}\ \mathit{Color}\ \mathit{Double}\ \mathit{Int} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{HasColor}\ a\ \textbf{where} & & \\
\quad \mathit{setColor} & :: & \mathit{Color} \to a \to a \\
\quad \mathit{setDefaultColor} & :: & a \to a \\
\quad \mathit{getColor} & :: & a \to \mathit{Color}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{HasBGColor}\ a\ \textbf{where} & & \\
\quad \mathit{setBGColor} & :: & \mathit{Color} \to a \to a \\
\quad \mathit{setDefaultBGColor} & :: & a \to a \\
\quad \mathit{getBGColor} & :: & a \to \mathit{Color}
\end{array}
$$

```
let rad n      = 60 * (1 / 1.2) ** n
    pol r a    = r *. dir (fromDouble a)
    color 0    = []
    color n    = areas (rad n) (λm → hsv2rgb (m, 1, 1)) ++ bw (n − 1)
    bw 0       = []
    bw n       = areas (rad n) (λm → grey (abs (m − 180) / 180)) ++ color (n − 1)
    areas      :: Numeric → (Double → Color) → [Area]
    areas r cf = [toArea [pol r i, pol r (2 + i),
                          pol (1.15 * r) (2 + i), pol (1.15 * r) i]
                   # setColor (cf i) # setPen 0.1
                 | i ← [0, 4 .. 356]]
in transform (affine (1, 0, 0.2, 0.85, 0, 0)) (color 9)
```

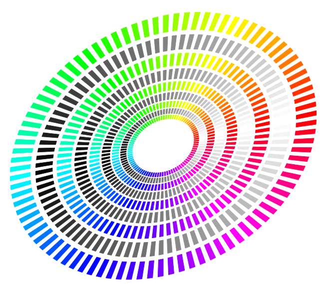
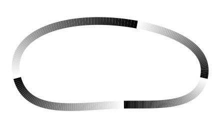


Figure 3.6: A color circle, see also [Ado85].

```
hsv2rgb                :: (Double, Double, Double) → Color
```

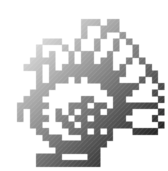
Here are two examples for color gradients:
– a color gradient on a path

```
(30, 0) .. (0, −10) .. (−40, 0) .. (5, 20) .. cycle
 # setColor (graduateMed white black 30)
 # setPen 3
```



– and a color gradient across a picture.

```
let turkey = ["003B00", "002700", "002480", "0E4940",
              "114920", "14B220", "3CB650", "75FE88",
              "17FF8C", "175F14", "1C07E2", "3803C4",
              "703182", "F8EDFC", "B2BBC2", "BB6F84",
              "31BFC2", "18EA3C", "0E3E00", "07FC00",
              "03F800", "1E1800", "1FF800"]
in scale 20 (image Depth1 turkey) # setColor (graduateMed white black 45)
```



$$
\begin{array}{lcl}
\mathit{graduate} & :: & \mathit{Color} \rightarrow \mathit{Color} \rightarrow \mathit{Double} \rightarrow \mathit{Int} \rightarrow \mathit{Color} \\
\mathit{graduate}\ c_1\ c_2\ a\ n & = & \mathit{Graduate}\ c_1\ c_2\ a\ n \\
\mathit{graduateLow} & :: & \mathit{Color} \rightarrow \mathit{Color} \rightarrow \mathit{Double} \rightarrow \mathit{Color} \\
\mathit{graduateLow}\ c_1\ c_2\ a & = & \mathit{graduate}\ c_1\ c_2\ a\ 16
\end{array}
$$

Figure 3.6 shows an application of the *hsv2rgb* function.

3.9 Dash patterns

Instances of *HasPattern*: *Frame*, *Path* and *PathElemDescr*.

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{HasPattern}\ a\ \textbf{where} & & \\
\quad \mathit{setPattern} & :: & \mathit{Pattern} \rightarrow a \rightarrow a \\
\quad \mathit{setDefaultPattern} & :: & a \rightarrow a \\
\quad \mathit{getPattern} & :: & a \rightarrow \mathit{Pattern}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Pattern} & = & \mathit{DefaultPattern} \\
 & \mid & \mathit{DashPattern}\ [\mathit{Double}] \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

3.10 Pencils

Instances of *HasPen*: *Frame*, *Path* and *PathElemDescr*.

Pen is Instance of: *Num* and *Fractional*

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{HasPen}\ a\ \textbf{where} & & \\
\quad \mathit{setPen} & :: & \mathit{Pen} \rightarrow a \rightarrow a \\
\quad \mathit{setDefaultPen} & :: & a \rightarrow a \\
\quad \mathit{getPen} & :: & a \rightarrow \mathit{Pen}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Pen} & = & \mathit{DefaultPen} \\
 & \mid & \mathit{PenSquare}\ (\mathit{Numeric}, \mathit{Numeric})\ \mathit{Numeric} \\
 & \mid & \mathit{PenCircle}\ (\mathit{Numeric}, \mathit{Numeric})\ \mathit{Numeric} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

3.11 Arrows

Instances of *HasArrowHead*: *Path* and *PathElemDescr*.

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{HasArrowHead}\ a\ \textbf{where} & & \\
\quad \mathit{setArrowHead} & :: & \mathit{ArrowHead} \rightarrow a \rightarrow a \\
\quad \mathit{removeArrowHead} & :: & a \rightarrow a \\
\quad \mathit{getArrowHead} & :: & a \rightarrow \mathit{Maybe}\ \mathit{ArrowHead} \\
\quad \mathit{setStartArrowHead} & :: & \mathit{ArrowHead} \rightarrow a \rightarrow a \\
\quad \mathit{removeStartArrowHead} & :: & a \rightarrow a \\
\quad \mathit{getStartArrowHead} & :: & a \rightarrow \mathit{Maybe}\ \mathit{ArrowHead}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{ArrowHead} & = & \mathit{DefaultArrowHead} \\
 & | & \mathit{ArrowHead}\ (\mathit{Maybe\ Double})\ (\mathit{Maybe\ Double}) \\
 & & \qquad\qquad \mathit{ArrowHeadStyle} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{ArrowHeadStyle} & = & \mathit{AHFilled} \\
 & | & \mathit{AHLine} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

3.12 Areas

Instances of *IsArea*: $\mathit{IsPath}\ a \Rightarrow \mathit{IsArea}\ [a]$, *Path* and *Area*.

Area is Instance of: *IsPicture*, *HasDefine*, *HasColor*, *HasPen*, *HasLayer*, *Show*, *Eq*, *Read*.

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{IsArea}\ a\ \textbf{where} & & \\
\quad \mathit{toArea} & :: & a \to \mathit{Area}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{HasLayer}\ a\ \textbf{where} & & \\
\quad \mathit{setBack} & :: & a \to a \\
\quad \mathit{setFront} & :: & a \to a \\
\quad \mathit{getLayer} & :: & a \to \mathit{Layer}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Area} & = & \mathit{Area\ AreaDescr\ Path} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{AreaDescr} & = & \mathit{AreaDescr}\{\mathit{arColor} :: \mathit{Color}, \\
 & & \qquad\qquad \mathit{arLayer} :: \mathit{Layer}, \\
 & & \qquad\qquad \mathit{arPen} :: \mathit{Pen}\} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Read})
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stdAreaDescr} & :: & \mathit{AreaDescr} \\
\mathit{stdAreaDescr} & = & \mathit{AreaDescr}\{\mathit{arColor} = \mathit{black}, \\
 & & \qquad\qquad \mathit{arLayer} = \mathit{Front}, \\
 & & \qquad\qquad \mathit{arPen} = \mathit{DefaultPen}\}
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Layer} & = & \mathit{Front}|\mathit{Back} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}
$$

3.13 Clipping

Clipping can be used for interesting effects, as demonstrated in figure 3.7. The bounding box of a *clip*ped picture is always rectangular.

```
let a = clip (ref NW -- ref NE -- ref SW -- cycle)
             (box "clip"
              # setBGColor (grey 0.8))
    b = clip (ref SW -- ref NE -- ref SE -- cycle)
             (box (tex "clip" # setColor white)
              # setBGColor black)
in column [a, vspace (−5), math "\\oplus", vspace (−5),
           b, math "\\downarrow", ooalign [a, b]]
```

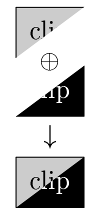


Figure 3.7: Clipping can be used for interesting effects.

3.14 Transformations

In addition to the transformations already mentioned we have

$$\begin{array}{lll} \mathit{reflectX}, \mathit{reflectY} & :: & \mathit{IsPicture}\ a \Rightarrow a \rightarrow \mathit{Picture} \end{array}$$

In order to apply several transformations to one picture it is more efficient to use

$$\begin{array}{lll} \mathit{transform} & :: & \mathit{IsPicture}\ a \Rightarrow \mathit{Transformation} \rightarrow a \rightarrow \mathit{Picture} \end{array}$$

and to combine the transformation matrices by the (&) operator. For example, a stretching along the x axis by a factor of two with a following rotation by 30 degree is achieved by the function *transform* (*scaledX* 2 & *rotated* 30). Some special transformation matrices are predefined and a general affine transformation can be constructed using *affine*.

$$\begin{array}{lll}
\mathit{shifted} & :: & (\mathit{Numeric}, \mathit{Numeric}) \rightarrow \mathit{Transformation} \\
\mathit{reflectedX}, \mathit{reflectedY} & :: & \mathit{Transformation} \\
\mathit{rotated}, \mathit{scaledX}, \mathit{scaledY}, & & \\
\mathit{scaled}, \mathit{skewedX}, \mathit{skewedY} & :: & \mathit{Numeric} \rightarrow \mathit{Transformation} \\
\mathit{affine} & :: & (\mathit{Numeric}, \mathit{Numeric}, \mathit{Numeric}, \mathit{Numeric}, \mathit{Numeric}, \mathit{Numeric}) \\
 & \rightarrow & \mathit{Transformation}
\end{array}$$

3.15 Bitmaps

$$\begin{array}{lll}
\mathbf{data}\ \mathit{BitDepth} & = & \mathit{Depth1} \mid \mathit{Depth8} \mid \mathit{Depth24} \\
 & & \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})
\end{array}$$

Figure 3.9 demonstrates the use of bitmaps in *functional* METAPOST.

$$\mathit{image} :: \mathit{BitDepth} \rightarrow [\mathit{String}] \rightarrow \mathit{Picture}$$

$$
\begin{array}{lll}
\textbf{let}\ \mathit{rek}\ 0\ \mathit{pic} & = & \mathit{pic} \\
\quad \mathit{rek}\ n\ \mathit{pic} & = & \mathit{ooalign}\ [\mathit{draw}\ [p]\ [\mathit{toArea}\ a\ \#\ \mathit{setColor}\ 0.6, \\
 & & \qquad\qquad\qquad \mathit{toArea}\ p\ \#\ \mathit{setColor}\ \mathit{white}], \\
 & & \qquad\qquad \mathit{rotate}\ 90\ (\mathit{scale}\ (1\ /\ 3)\ (\mathit{rek}\ (n-1)\ \mathit{pic}))] \\
\quad p & = & \mathit{transformPath}\ (\mathit{scaled}\ 30)\ \mathit{fullcircle} \\
\quad a & = & (\mathit{vec}\ (90,0)\ ..\ \mathit{vec}\ (0,30)\ ..\ \mathit{vec}\ (-90,0)\ \#\ \mathit{setEndCurl}\ 1) \\
 & & \quad ..\ \mathit{vec}\ (0,-30)\ ..\ \mathit{cycle}\ \#\ \mathit{setEndCurl}\ 1 \\
\textbf{in}\ \mathit{rek}\ 6\ \mathit{empty} & &
\end{array}
$$

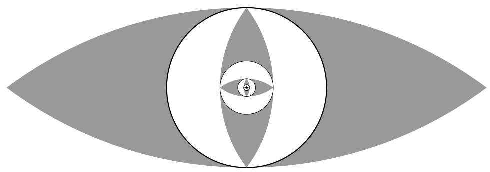


Figure 3.8: A recursive picture (See figure 28 in [Hob92]).

$$
\begin{array}{l}
\mathit{rowSepBy}\ 10\ [\mathit{rbox}\ 15\ (\mathit{scale}\ 3\ (\mathit{image}\ \mathit{Depth24}\ \mathit{fruits}))\ \#\ \mathit{setDX}\ 10\ \#\ \mathit{setDY}\ 10, \\
\qquad\qquad\quad \mathit{rbox}\ 15\ (\mathit{scale}\ 6\ (\mathit{image}\ \mathit{Depth8}\ \mathit{tiger}))\ \#\ \mathit{setDX}\ 10\ \#\ \mathit{setDY}\ 10, \\
\qquad\qquad\quad \mathit{fuzzy}\ 4\ 5\ (\mathit{scale}\ 2\ (\mathit{image}\ \mathit{Depth1}\ \mathit{woodpecker}))]
\end{array}
$$

Figure 3.9: Three bitmaps with color depths 24, 8 and one bit.

3.16 Extensions

3.16.1 Canvas graphics

Canvas is Instance of: *IsPicture*, *HasRelax* and *HasConcat*.

3.16.2 Turtle graphics

Turtle is Instance of: *IsPicture*, *IsHideable*, *HasRelax* and *HasConcat* *HasPicture*, *HasColor*, *HasPen*.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Turtle} & = & \mathit{TConc\ Turtle\ Turtle} \\
 & \mid & \mathit{TDropPic\ Picture} \\
 & \mid & \mathit{TColor\ Color\ Turtle} \\
 & \mid & \mathit{TPen\ Pen\ Turtle} \\
 & \mid & \mathit{THide\ Turtle} \\
 & \mid & \mathit{TForward\ Numeric} \\
 & \mid & \mathit{TTurn\ Numeric} \\
 & \mid & \mathit{TPenUp} \\
 & \mid & \mathit{TPenDown} \\
 & \mid & \mathit{THome} \\
 & \mid & \mathit{TFork\ Turtle\ Turtle} \\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

It is useful to have functions where the rotation direction (for positive arguments) is part of the name:

$$
\begin{array}{lcl}
\mathit{turnl} & :: & \mathit{Numeric} \rightarrow \mathit{Turtle} \\
\mathit{turnl}\ a & = & \mathit{TTurn}\ a \\
\mathit{turnr} & :: & \mathit{Numeric} \rightarrow \mathit{Turtle} \\
\mathit{turnr}\ a & = & \mathit{TTurn}\ (-a)
\end{array}
$$

3.16.3 Trees

Tree is Instance of: *IsPicture* and *HasName*.

Edge is Instance of: *HasColor*, *HasLabel*, *HasPen*, *HasPattern*, *HasArrowHead* *HasStartEndDir*, *IsHideable*

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree} & = & \mathit{Node\ Picture\ NodeDescr}\ [\mathit{Edge}] \\
 & & \textbf{deriving}\ \mathit{Show} \\
\textbf{data}\ \mathit{Edge} & = & \mathit{Edge\ Path\ Tree} \\
 & \mid & \mathit{Cross\ Path} \\
 & & \textbf{deriving}\ \mathit{Show} \\
\textbf{data}\ \mathit{NodeDescr} & = & \mathit{NodeDescr}\{\mathit{nEdges} :: [\mathit{Path}], \\
 & & \qquad \mathit{nAlignSons} :: \mathit{AlignSons}, \\
 & & \qquad \mathit{nDistH}, \mathit{nDistV} :: \mathit{Distance}\} \\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{stdNodeDescr} & :: & \mathit{NodeDescr} \\
\mathit{stdNodeDescr} & = & \mathit{NodeDescr}\{\,\mathit{nEdges} = [\,], \\
 & & \qquad\qquad\quad \mathit{nAlignSons} = \mathit{DefaultAlign}, \\
 & & \qquad\qquad\quad \mathit{nDistH} = 8, \\
 & & \qquad\qquad\quad \mathit{nDistV} = 10\,\}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Distance} & = & \mathit{DistCenter}\ \mathit{Numeric} \\
 & \mid & \mathit{DistBorder}\ \mathit{Numeric} \\
 & & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{array}
$$

$$
\begin{array}{lll}
\textbf{data}\ \mathit{NodeName} & = & \mathit{Parent} \mid \mathit{This} \mid \mathit{Root} \mid \mathit{Up}\ \mathit{Int} \mid \mathit{Son}\ \mathit{Int} \\
 & & \textbf{deriving}\ \mathit{Show}
\end{array}
$$

There are a number of functions to apply attribute functions to a whole tree. You may apply an attribute to all nodes, to all nodes of a given level, to all pictures or to all edges:

$$
\begin{array}{lll}
\mathit{forEachNode} & :: & (\mathit{Tree} \to \mathit{Tree}) \to \mathit{Tree} \to \mathit{Tree} \\
\mathit{forEachLevelNode} & :: & \mathit{Int} \to (\mathit{Tree} \to \mathit{Tree}) \to \mathit{Tree} \to \mathit{Tree} \\
\mathit{forEachPic} & :: & (\mathit{Picture} \to \mathit{Picture}) \to \mathit{Tree} \to \mathit{Tree} \\
\mathit{forEachEdge} & :: & (\mathit{Path} \to \mathit{Path}) \to \mathit{Tree} \to \mathit{Tree}
\end{array}
$$

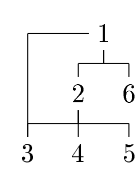
The following function can save some place in a tree description:

$$
\begin{array}{lll}
\mathit{enode} & :: & \mathit{IsPicture}\ a \Rightarrow a \to [\mathit{Edge}] \to \mathit{Edge} \\
\mathit{enode}\ p\ \mathit{ts} & = & \mathit{edge}\ (\mathit{node}\ p\ \mathit{ts})
\end{array}
$$

The drawing of edges can be redefined using arbitrary paths. Here is an example using stair like edges:

$$
\begin{array}{lll}
\mathit{cross}' & :: & \mathit{Path} \to \mathit{Edge} \\
\mathit{cross}' & = & \mathit{Cross} \\[4pt]
\mathit{edge}' & :: & \mathit{Path} \to \mathit{Tree} \to \mathit{Edge} \\
\mathit{edge}' & = & \mathit{Edge} \\[4pt]
\mathit{stair} & :: & \mathit{Point} \to \mathit{Point} \to \mathit{Path} \\
\mathit{stair}\ p_1\ p_2 & = & p_1\ \text{--}\ p_1 + \mathit{vec}\ (0, 0.5 * \mathit{ydist}\ p_2\ p_1) \\
 & & \text{--}p_2 - \mathit{vec}\ (0, 0.5 * \mathit{ydist}\ p_2\ p_1)\ \text{--}\ p_2
\end{array}
$$

$$
\begin{array}{l}
\textbf{let}\ \mathit{sedge} = \mathit{edge}'\ (\mathit{stair}\ (\mathit{ref}\ (\mathit{This} \triangleleft C))\ (\mathit{ref}\ (\mathit{Parent} \triangleleft C))) \\
\quad\ \ \mathit{scross}\ p = \mathit{cross}'\ ((\mathit{ref}\ (\mathit{This} \triangleleft C))\ \text{--}\ \mathit{xy}\ p\ (\mathit{ref}\ (\mathit{This} \triangleleft C))\ \text{--}\ p) \\
\textbf{in}\ \mathit{node}\ \texttt{"1"}\ [\mathit{sedge}\ (\mathit{node}\ \texttt{"2"}\ [\mathit{sedge}\ (\mathit{node}\ \texttt{"3"}\ [\,]\ \#\ \mathit{setName}\ \texttt{"3"}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{sedge}\ (\mathit{node}\ \texttt{"4"}\ [\,]), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{sedge}\ (\mathit{node}\ \texttt{"5"}\ [\,])]), \\
\qquad\qquad\quad \mathit{sedge}\ (\mathit{node}\ \texttt{"6"}\ [\,]), \\
\qquad\qquad\quad \mathit{scross}\ (\mathit{ref}\ (\texttt{"3"} \triangleleft C))]
\end{array}
$$



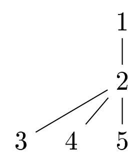
For the alignment of the children of a node exist eight predefined options and the possibility to define an own alignment algorithm.

```
data AlignSons          = DefaultAlign
                        | AlignLeft
                        | AlignRight
                        | AlignLeftSon
                        | AlignRightSon
                        | AlignOverN Int
                        | AlignAngles [Numeric]
                        | AlignConst Numeric
                        | AlignFunction (NodeDescr → [Extent] → Int → [Numeric])
                          deriving Show
```

It is possible to draw the parent node above the nth child node:

```
alignOverN              :: Int → AlignSons
```

```
node "1" [edge (node "2" [edge (node "3" []),
                          edge (node "4" []),
                          edge (node "5" [])]
                    # setAlign (alignOverN 3))]
```

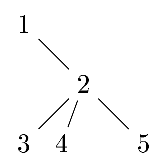


It is also possible to prescibe edge angles.

```
alignAngles             :: [Double] → AlignSons
```

The edges (1,2) and (2,5) lie on one line in the following example. This would also work for quite different horizontal distances between the levels.

```
node "1" [edge (node "2" [edge (node "3" []),
                          edge (node "4" []),
                          edge (node "5" [])]
                    # setAlign (alignAngles [45, 70, −45]))]
  # setAlign (alignAngles [−45])
```

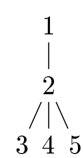


On the other way, it is also possible to prescribe the horizontal distance:

```
alignConst              :: Double → AlignSons
```

The parent gets centered over its child nodes

```
node "1" [edge (node "2" [edge (node "3" []),
                          edge (node "4" []),
                          edge (node "5" [])]
                    # setAlign (alignConst 10))]
```



It is possible to define the alignment algorithm oneself. The first parameter of type *NodeDescr* allows access to the attributes, such as desired node distances. The list of type [*Extend*] has all the outlines of the subtrees. The last parameter is the level in the tree. Alignment can depend on it, too. The function must return a list of numbers, which describe for every subtree the position relative to the parent node.

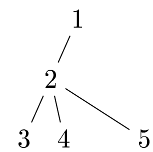
$$\textit{alignFunction} \quad :: \quad (\textit{NodeDescr} \to [\textit{Extent}] \to \textit{Int} \to [\textit{Numeric}]) \to \textit{AlignSons}$$

One trivial example is to simply let the user prescripe these relative positions:

$$\textit{alignConsts}\ cs \quad = \quad \textit{alignFunction}\ (\lambda\_\ \_\ \_ \to cs)$$

The outlines of subtrees are not taken into consideration here. Therefore it may happen that the subtrees will overlap.

$$\begin{array}{l} \textit{node}\ \texttt{"1"}\ [\textit{edge}\ (\textit{node}\ \texttt{"2"}\ [\textit{edge}\ (\textit{node}\ \texttt{"3"}\ [\,]), \\ \qquad\qquad\qquad\qquad\qquad\quad \textit{edge}\ (\textit{node}\ \texttt{"4"}\ [\,]), \\ \qquad\qquad\qquad\qquad\qquad\quad \textit{edge}\ (\textit{node}\ \texttt{"5"}\ [\,])] \\ \qquad\qquad\qquad \#\ \textit{setAlign}\ (\textit{alignConsts}\ [-10, 5, 35]))] \\ \#\ \textit{setAlign}\ (\textit{alignConsts}\ [-10]) \end{array}$$



This simple definition leads to an error if there are more subtrees as numbers in the list. A more robust variant which extends the list in such cases is:

$$\begin{array}{lcl} \textit{alignConsts}'\ cs & = & \textit{alignFunction}\ (\lambda\_\ es\ \_ \\ & & \qquad \to \mathbf{let}\ n = \textit{length}\ es \\ & & \qquad\quad \mathbf{in}\ \mathbf{if}\ n > \textit{length}\ cs \\ & & \qquad\qquad \mathbf{then}\ \textit{resumeList}\ n\ cs \\ & & \qquad\qquad \mathbf{else}\ cs) \\ \quad \mathbf{where} & & \\ \quad \textit{resumeList}\ n\ [\,] & = & [0] \\ \quad \textit{resumeList}\ n\ [c] & = & \textit{take}\ n\ [c, 2 * c\,..] \\ \quad \textit{resumeList}\ n\ cs & = & \textit{take}\ n\ (cs +\!\!+ [\textit{last}\ cs + d, \textit{last}\ cs + 2 * d\,..]) \\ \qquad \mathbf{where} & & \\ \qquad d & = & \textit{last}\ cs - \textit{last}\ (\textit{init}\ cs) \end{array}$$

Appendix A

ASCII representation of operators

For better readability some operators are pretty-printed in this paper. This is done automatically using the `lhs2TeX` program by Ralf Hinze.

<table>
<tr><th>Pretty Printed</th><th>ASCII</th><th></th></tr>
<tr><td>$\square\!\square$</td><td><code>|||</code></td><td rowspan="4">Alignements</td></tr>
<tr><td>$\square\,\square$</td><td><code>||||</code></td></tr>
<tr><td>$\overset{\square}{\square}$</td><td><code>|-|</code></td></tr>
<tr><td>$\overset{\square}{\underset{}{\square}}$</td><td><code>|=|</code></td></tr>
<tr><td>--</td><td><code>.-.</code></td><td rowspan="4">Path constructors</td></tr>
<tr><td>---</td><td><code>.--.</code></td></tr>
<tr><td>..</td><td><code>...</code></td></tr>
<tr><td>...</td><td><code>....</code></td></tr>
<tr><td>$\triangleleft$</td><td><code><+</code></td><td rowspan="2">Name constructors</td></tr>
<tr><td>$\triangleleft\!\!*$</td><td><code><*</code></td></tr>
<tr><td>$\doteq$</td><td><code>.=</code></td><td>Equality</td></tr>
<tr><td>$\dot{<}$</td><td><code>.<</code></td><td rowspan="4">Comparisons</td></tr>
<tr><td>$\dot{\leqslant}$</td><td><code>.<=</code></td></tr>
<tr><td>$\dot{\equiv}$</td><td><code>.==</code></td></tr>
<tr><td>$\dot{\not\equiv}$</td><td><code>./=</code></td></tr>
<tr><td>$\dot{*}$</td><td><code>.*</code></td><td>Multiplication Number – Point</td></tr>
<tr><td>$\langle\!+\!\rangle$</td><td><code><+></code></td><td rowspan="2">Pretty printer</td></tr>
<tr><td>$\$\!\$$</td><td><code>$$</code></td></tr>
<tr><td>λ</td><td><code>\</code></td><td>Lambda Abstraktion</td></tr>
<tr><td>$cycle$</td><td><code>cycle'</code></td><td rowspan="2"></td></tr>
<tr><td>$default$</td><td><code>default'</code></td></tr>
</table>

Bibliography

[Abe85] H. Abelson. *Einführung in LOGO.* IWT-Verlag, München, second edition, 1985.

[Ad82] H. Abelson and A. A. diSessa. *TurtleGeometry.* MIT Press, third edition, 1982.

[Ado85] Adobe Systems, Inc. *PostScript® Language Reference Manual.* Adobe Systems Incorporated, 1985.

[Ado88] Adobe Systems, Inc. *PostScript® Language Program Design.* Adobe Systems Incorporated, 1988.

[BO96] Gerth Stølting Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, November 1996.

[Fel92] W. D. Fellner. *Computergrafik.* BI-Wissenschaftsverlag, 1992.

[GKP92] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, MA, USA, eigth edition, 1992.

[Hob92] J. D. Hobby A user's manual for METAPOST. Computing Science Technical Report no. 162, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992.

[Man87] Benoît B. Mandelbrot. *Die fraktale Geometrie der Natur.* Birkhäuser, Basel, 1987.

[Oka98] Chris Okasaki. Functional pearls: Constructing red-black trees in a functional setting. *Journal of functional programming*, 1998. to appear.

[Pap82] Seymour Papert. *Mindstorms: Kinder, Computer und neues Lernen.* Birkhäuser Verlag, Basel—Boston—Stuttgart, 1982.

[Sch97] Uwe Schöning. *Theoretische Informatik kurz gefasst.* Spektrum, third edition, 1997.

[Sed92] Robert Sedgewick. *Algorithmen.* Addison-Wesley Verlag, Bonn, Germany, 1992.

Index