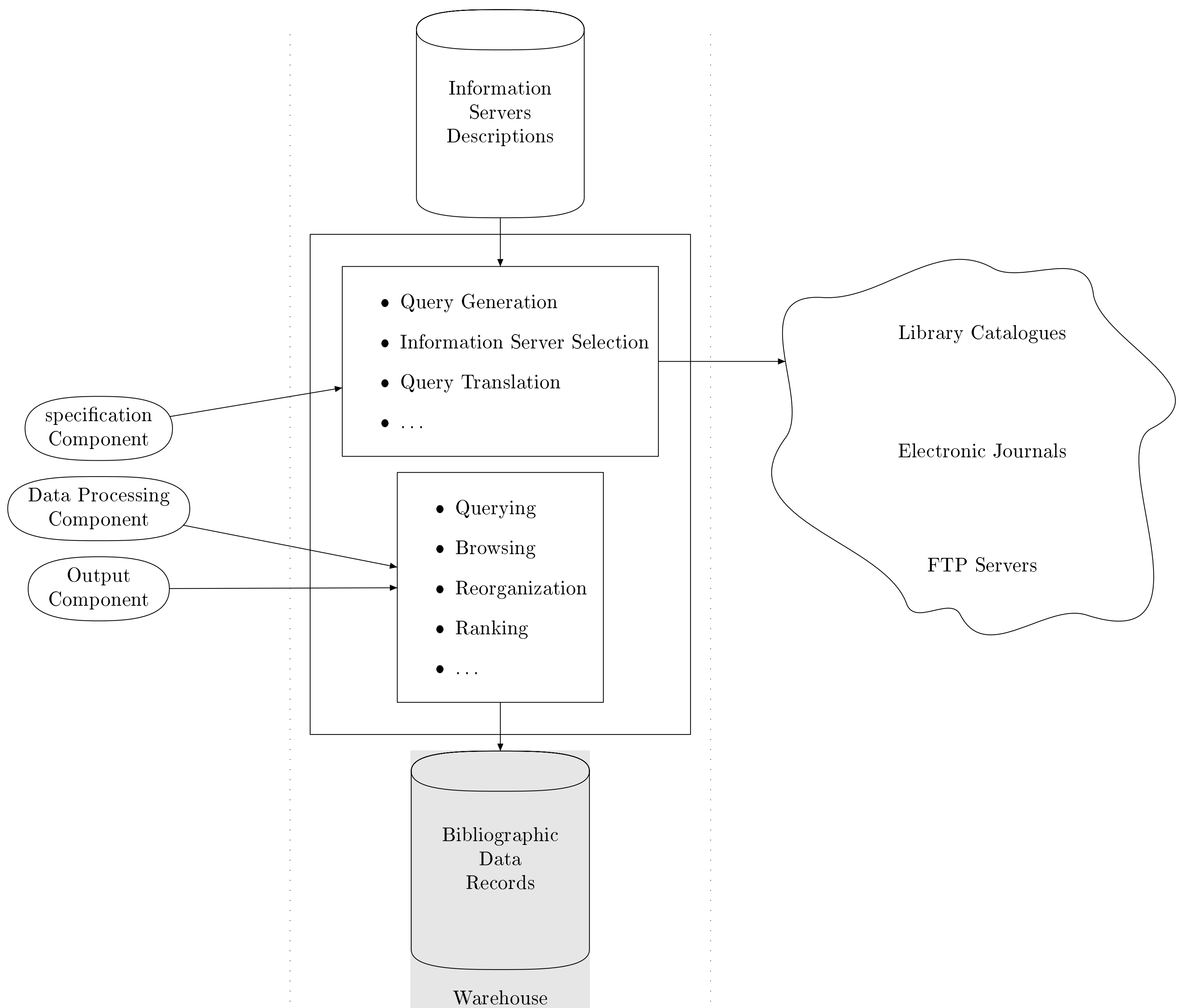
# *functional* METAPOST
# Eine Beschreibungssprache für Grafiken
# Tutorial

Joachim Korittky

15. November 1998

## Inhaltsverzeichnis

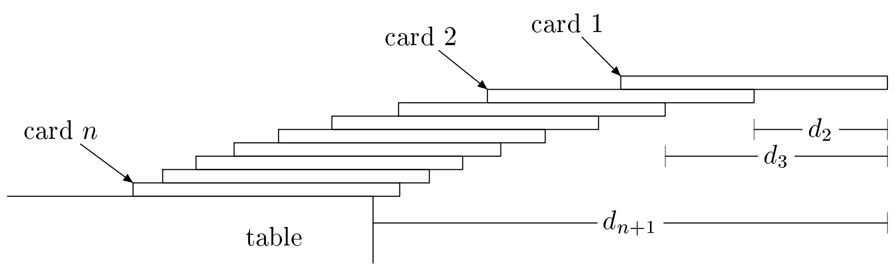
```
dbs                    = draw [arrow ("Information" ◁ C) ("Query" ◁ C),
                                 arrow ("Spec" ◁ C) ("Query" ◁ C),
                                 arrow ("Proc" ◁ C) ("Querying" ◁ C),
                                 arrow ("Output" ◁ C) ("Querying" ◁ C),
                                 arrow ("Querying" ◁ C) ("Bib" ◁ C),
                                 arrow ("Query" ◁ C) (xy (ref ("Servers" ◁ C)) (ref ("Query" ◁ C)))
                                       # setEndCut "Servers",
                                 line (xy (ref ("Center" ◁ W) − vec (10,0)) (ref ("Information" ◁ N)))
                                      (xy (ref ("Center" ◁ W) − vec (10,0)) (ref ("Warehouse" ◁ S)))
                                       # setPattern dotted,
                                 line (xy (ref ("Center" ◁ E) + vec (10,0)) (ref ("Information" ◁ N)))
                                      (xy (ref ("Center" ◁ E) + vec (10,0)) (ref ("Warehouse" ◁ S)))
                                       # setPattern dotted
                                 ]
                         (cdrop (0,0) frontEnd & cdrop (200,0) backEnd
                       &  cdrop (440,30) servers)
    where
    frontEnd           = (oval (texCol "specification\\\\Component") # setName "Spec")
                       ⊟ (oval (texCol "Data␣Processing\\\\Component") # setName "Proc")
                       ⊟ (oval (texCol "Output\\\\Component") # setName "Output")
    backEnd            =  setTrueBoundingBox (drum (texCol "Information\\\\Servers\\\\Descriptions") # setName "Information")
                       ⊟ (box ((box "\\parbox{150pt}{\\begin{itemize}\\item␣Query␣Generation\\item␣Information␣Server␣Selection\\item␣Query␣Translation\\item$\\ldots$\\end{itemize}}" # setName "Query")
                             ⊟ (box "\\parbox{95pt}{\\begin{itemize}\\item␣Querying\\item␣Browsing\\item␣Reorganization\\item␣Ranking\\item$\\ldots$\\end{itemize}}" # setName "Querying")) # setName "Center" # setDX 16 # setDY 16)
                       ⊟ (setTrueBoundingBox (drum (texCol "Bibliographic\\\\Data\\\\Records") # setName "Bib")
                             ⊟ "Warehouse" # setBGColor 0.9 # setName "Warehouse")
    servers            = fuzzy 1 2 (tex "Library␣Catalogues" ⊟ vspace 30
                             ⊟ tex "Electronic␣Journals" ⊟ vspace 30
                             ⊟ tex "FTP␣Servers") # setName "Servers"
    texCol c           = tex ("\\begin{tabular}{c}" ++ c ++ "\\end{tabular}")
```



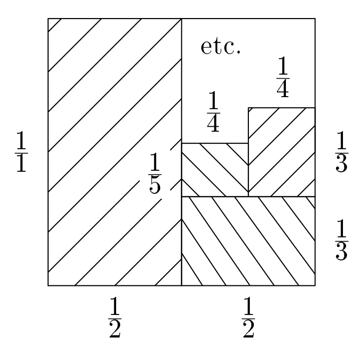
Information
Servers
Descriptions
Query Generation
Information Server Selection
Query Translation
. . .
specification
Component
Data Processing
Component
Output
Component
Querying
Browsing
Reorganization
Ranking
. . .
Library Catalogues
Electronic Journals
FTP Servers
Bibliographic
Data
Records
Warehouse

## 0.1 Einführung

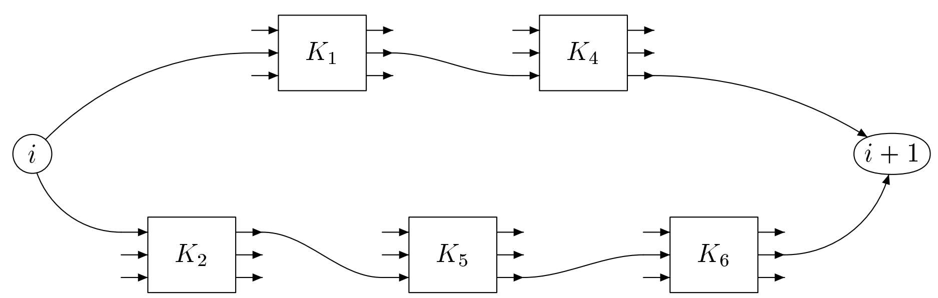
```
conc259        = draw [arrow ("1" ◁ C) (dk !! 0, 0),
                       arrow ("2" ◁ C) (dk !! 1, −5),
                       arrow ("n" ◁ C) (dk !! 8, −40)]
                 (cards
               & cdraw (darrow (100 + dk !! 1, −20) (0, −20) "d_2")
               & cdraw (darrow (100 + dk !! 2, −30) (0, −30) "d_3")
               & cdraw (darrow (100 − 100 ∗ d 11, −55) (0, −55) "d_{n+1}")
               & cdraw ((−330, −45) -- (100 − 100 ∗ d 11, −45)
                              -- (100 − 100 ∗ d 11, −70))
               & cdrop (−230, −60) "table"
               & cdrop (−120, 20) (tex "card␣$1$" # setName "1")
               & cdrop (−175, 15) (tex "card␣$2$" # setName "2")
               & cdrop (−310, −20) (tex "card␣$n$" # setName "n"))
    where
    cards        = foldl ( & ) relax [cdraw ((x, y) -- (x + 100, y)
                        -- (x + 100, y − 5) -- (x, y − 5) -- cycle′)
                              | (x, y) ← zip dk [0, −5 . .]]
    dk           = [−100 ∗ d i | i ← [2 . . 10]]
    d 1          = 0
    d n          = (n − 1 + sum [d i | i ← [1 . . (n − 1)]]) / (n − 1)
    darrow a b s = a -- b # setArrowHead (arrowHeadSize 4 180)
                        # setStartArrowHead (arrowHeadSize 4 180)
                        # setLabel 0.5 C (math s # setBGColor white)
```



$$
\begin{array}{lll}
\mathit{conc66} & = & \mathit{box}\ (0,0)\ (50,100)\ (\lambda i \to (0, 200 - i * 1.5)) \\
 & & \qquad\qquad\qquad\qquad\quad (\lambda i \to (i * 1.5, 200)) \\
 & \& & \mathit{box}\ (50,0)\ (100,33.3)\ (\lambda i \to (0,i))\ (\lambda i \to (i,-50)) \\
 & \& & \mathit{box}\ (75,33.3)\ (100,66.6)\ (\lambda i \to (0,100-i))\ (\lambda i \to (i,100)) \\
 & \& & \mathit{box}\ (50,33.3)\ (75,53.3)\ (\lambda i \to (0,i))\ (\lambda i \to (i,0)) \\
 & \& & \mathit{cdraw}\ (\mathit{vec}\ (50,100)\ \text{--}\ \mathit{vec}\ (100,100)\ \text{--}\ \mathit{vec}\ (100,50)) \\
 & \& & \mathit{frac}\ (-10,50)\ 1 \\
 & \& & \mathit{frac}\ (25,-12)\ 2 \\
 & \& & \mathit{frac}\ (75,-12)\ 2 \\
 & \& & \mathit{frac}\ (110,17)\ 3 \\
 & \& & \mathit{frac}\ (110,50)\ 3 \\
 & \& & \mathit{frac}\ (62,65)\ 4 \\
 & \& & \mathit{frac}\ (88,78)\ 4 \\
 & \& & \mathit{frac}\ (40,42)\ 5 \\
 & \& & \mathit{cdrop}\ (65,90)\ \texttt{"etc."} \\
\quad \textbf{where} & & \\
\quad \mathit{frac}\ p\ n & = & \mathit{cdrop}\ p\ (\mathit{math}\ (\texttt{"\\\\frac\{\\\\textstyle␣1\}\{\\\\textstyle␣"} \\
 & & \qquad\qquad +\!\!+ \mathit{show}\ n +\!\!+ \texttt{"\}"})\ \#\ \mathit{setBGColor\ white}) \\
\quad \mathit{box}\ (ax,ay)\ (bx,by)\ \mathit{fa}\ \mathit{fb} & = & \mathit{cdrop}\ (0.5 * (ax + bx), 0.5 * (ay + by)) \\
 & & \qquad (\mathit{cdraws}\ [\mathit{fa}\ i\ \text{--}\ \mathit{fb}\ i \mid i \leftarrow [-50,-40\,..\,200]] \\
 & & \qquad \ \& \ \mathit{cclip}\ p \\
 & & \qquad \ \& \ \mathit{cdraw}\ p) \\
\qquad \textbf{where} & & \\
\qquad p & = & \mathit{vec}\ (ax,ay)\ \text{--}\ \mathit{vec}\ (bx,ay)\ \text{--}\ \mathit{vec}\ (bx,by) \\
 & & \ \text{--}\ \mathit{vec}\ (ax,by)\ \text{--}\ \mathit{cycle}'
\end{array}
$$

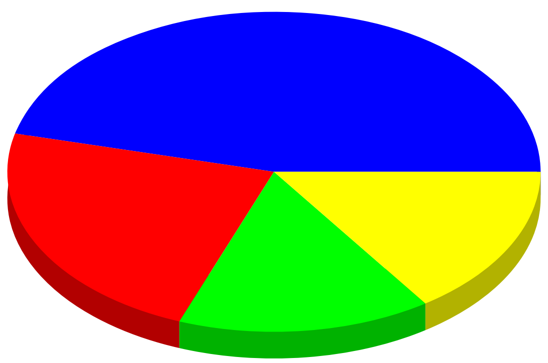


```
hamilton          = arrows (matrix [[empty, empty, k 1, empty, k 4],
                                    [i ◫ hspace 20, empty, empty, empty,
                                     empty, empty, hspace 20 ◫ ip1],
                                    [empty, k 2, empty, k 5, empty,
                                     k 6 # setName "6"]])
    where
    i             = circle (math "i") # setName "i"
    ip1           = oval (math "i+1") # setName "ip1"
    k s           = boxArrows s (box (math ("K_" ++ show s))
                     # setDX 8
                     # setDY 8)
    arrows        = draw [("i" ◁ C) .. (2◁1◁C) # setEndAngle 0,
                          ("i" ◁ C) .. (1◁2◁C) # setEndAngle 0,
                          curve (1◁5◁C) 0 0 (4◁3◁C),
                          curve (2◁4◁C) 0 0 (5◁3◁C),
                          curve (5◁6◁C) 0 0 (6◁2◁C),
                          arrow (4◁6◁C) ("ip1" ◁ C) # setStartAngle 0,
                          arrow (6◁5◁C) ("ip1" ◁ C) # setStartAngle 0]
    curve x a1 a2 y = x .. y # setStartAngle a1 # setEndAngle a2
    boxArrows     :: Int → Frame → Picture
    boxArrows s   = setName s ∘ draw [out 4 0.2, out 5 0.5, out 6 0.8,
                                      to 1 0.2, to 2 0.5, to 3 0.8]
        where
        out m n   = arrow (med n (ref NE) (ref SE))
                          (med n (ref NE) (ref SE) + vec (10, 0))
                     # setLabel 1 C (empty # setName (m :: Int))
        to m n    = arrow (med n (ref NW) (ref SW) − vec (10, 0))
                          (med n (ref NW) (ref SW))
                     # setLabel 0 C (empty # setName (m :: Int))
```



```
torte = cake [(30, blue), (15, red), (10, green), (10, yellow)]
  where
  cake as = drawBack (toAngles as)
          & drawCake (toAngles as)
  toAngles as = ad (map (λ   (   a, c) → (a / (sum (map fst as) / 360), c)) as)
  ad [] = [(0, black)]
  ad ((a, c) : as) = (a + fst (head (ad as)), c) : ad as
  drawBack [] = relax
  drawBack [_] = relax
  drawBack ((a, c) : (b, c2) : as)
                  = cfill   (   toArea (carc' (vec (0, −10)) 100 60 a b -- carc' (vec (0, 0)) 100 40 b a -- cycle')
                            #   setColor (c − grey 0.3)) & drawBack ((b, c2) : as)
  drawCake [] = relax
  drawCake [_] = relax
  drawCake ((a, c) : (b, c2) : as)
                  = cfill (toArea (vec (0, 0) -- carc' (vec (0, 0)) 100 60 a b -- cycle')
                            #   setColor c) & drawCake ((b, c2) : as)

carc' :: Point → Numeric → Numeric → Numeric → Numeric → Path
carc' p rx ry w1 w2          =   (p + vec (rx ∗ cos a1, −ry ∗ sin a1)
                             ..   p + vec (rx ∗ cos a2, −ry ∗ sin a2)
                             ..   p + vec (rx ∗ cos a2b, −ry ∗ sin a2b)
                             ..   p + vec (rx ∗ cos a3, −ry ∗ sin a3)
                             ..   p + vec (rx ∗ cos a4b, −ry ∗ sin a4b)
                             ..   p + vec (rx ∗ cos a4, −ry ∗ sin a4)
                             ..   p + vec (rx ∗ cos a5, −ry ∗ sin a5))
       where
       a1                    =   w1
       a2                    =   w1 + 0.05 ∗ (w2 − w1)
       a2b                   =   w1 + 0.25 ∗ (w2 − w1)
       a3                    =   w1 + 0.5 ∗ (w2 − w1)
       a4b                   =   w1 + 0.75 ∗ (w2 − w1)
       a4                    =   w1 + 0.95 ∗ (w2 − w1)
       a5                    =   w2
```



## 0.2 Haskell

eingebettete Sprache Klassen, Pragmatics/Default etc. generelle Idee

## 0.3 Atomare Bilder

Der Grundtyp auf dem die Sprache *functional* METAPOST aufbaut ist *Picture* bzw. kürzer *Pic*. Mit Ausdrücken dieses Typs kann man Bilder beschreiben, die sich dann in ein METAPOST–Programm kompilieren oder zu neuen, komplizierteren Bildern kombinieren lassen.
Mit atomaren Bildern sind hier solche gemeint, die nicht aus Kombination oder Veränderung anderer hervorgehen.
Zwei solche atomare Funktionen sind die Einbettung von beliebigen LaTeX–Ausdrücken [1] und das Erzeugen eines rechteckigen, leeren Bildes bestimmter Größe.

$$\begin{aligned} &tex && :: String \to Picture \\ &space && :: Numeric \to Numeric \to Picture \end{aligned}$$

Der Zweck der Funktion *space* ist vielleicht nicht sofort klar, da nichts sichtbares erzeugt wird. Vielmehr liefert die Funktion einen Platzhalter, denn jedes Bild hat eine es umgebende bounding box [2]. Anhand der bounding boxen lassen sich Bilder ohne Überlappungen zu komplizierteren Bildern kombinieren.
Leicht ergeben sich aus den beiden atomaren Funktionen *tex* und *space* weitere nützliche Funktionen. Z.B. zum automatischen Satz im Mathematikmodus von LaTeX

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

oder in Anlehnung an die entsprechenden Befehle in LaTeX zum Erzeugen horizontaler und vertikaler leerräume

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

Eine Funktion, die wie sich noch zeigen wird auch nützlich sein kann, erzeugt das leere Bild.

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

## 0.4 Farben

Jedes sichtbare Objekt in *functional* METAPOST kann eine Farbe erhalten. Die Voreinstellung ist dabei *black*. Als Farbraum kommt das RGB-Modell zur Anwendung, bei dem eine Farbe durch additives Mischen der Grundfarben rot grün und blau beschrieben wird.

$$color \quad :: Double \to Double \to Double \to Color$$

Es stehen schon Funktionen für häufig benötigte Farben zur Verfügung.

$$\begin{aligned} &white && = color\ 1\ 1\ 1 \\ &black && = color\ 0\ 0\ 0 \\ &red && = color\ 1\ 0\ 0 \\ &green && = color\ 0\ 1\ 0 \\ &blue && = color\ 0\ 0\ 1 \end{aligned}$$

[1] Zu beachten ist der Vorteil der Verwendung von LaTeX in den Bildern, wenn der umgebende Text im Dokument auch mit LaTeX gesetzt ist. Da die gleichen Zeichensätzt und Formatierungen zur Anwendung kommen, paßt sich die Erscheinungs des Bildes sehr gut in den Text ein. Die volle Mächtigkeit von LaTeX steht zur Verfügung.

[2] Die bounding box muß nicht wirklich rechteckig, sondern kann auch rund oder dreieckig sein.

$$
\begin{array}{lcl}
\mathit{yellow} & = & \mathit{color}\ 1\ 1\ 0 \\
\mathit{cyan} & = & \mathit{color}\ 0\ 1\ 1 \\
\mathit{magenta} & = & \mathit{color}\ 1\ 0\ 1 \\
\mathit{grey}\ n & = & \mathit{color}\ n\ n\ n
\end{array}
$$

Ferner ist der Typ *Color* eine Instanz von *Num* und *Fractional*, was bedeutet, daß sich Farben auch addieren, subtrahieren sowie multiplizieren lassen. Diese Operationen erfolgen komponentenweise auf den roten, grünen bzw. blauen Anteilen der Farben. Der Ausdruck $\mathit{red} + \mathit{green}$ bezeichnet also die Farbe *yellow* und $\mathit{cyan} - \mathit{blue}$ bezeichnet die Farbe *green*. Weil der Typ *Color* auch die Funktion *fromRational* implemmentiert, steht $0.5 * \mathit{magenta}$ für $\mathit{grey}\ 0.5 * \mathit{magenta}$, d.h. $\mathit{color}\ 0.5\ 0\ 0.5$.
Um in einem Bild den Farbwert zu ändern, existiert die Funktion

$$\mathit{setColor} \quad :: \quad \mathit{Color} \to \mathit{Picture} \to \mathit{Picture}$$

Eigenschaften von Bildern wie Farben kann man als Attribute betrachten, die einen bestimmten voreingestellten Wert haben. Wir wollen die Attributmenge von Objekten als Attributierung und die Funktionen diese zu verändern als Attributierungsfunktionen bezeichnen. Eine Attributierungsfunktion zum Setzen der Hintergrundfarbe ist

$$\mathit{setBGColor} \quad :: \quad \mathit{Color} \to \mathit{Picture} \to \mathit{Picture}$$

Attributierungen lassen sich besser lesen, wenn man die postfix-Notation verwendet. Dazu dient der # - Operator.

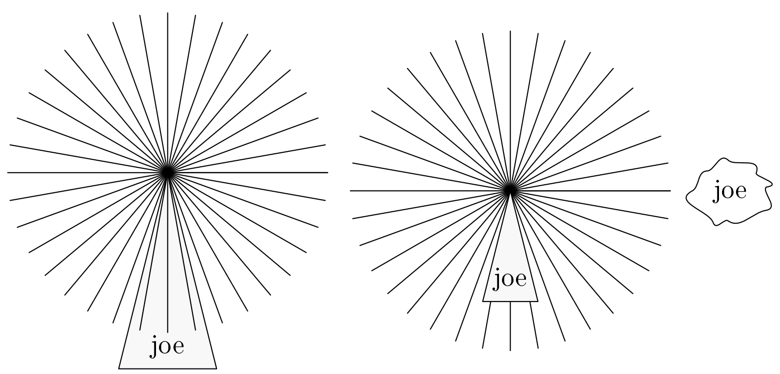
$$
\begin{array}{lcl}
(\,\#\,) & :: & a \to (a \to b) \to b \\
a\ \#\ f & = & f\ a
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{color1} & = & \mathit{tex}\ \texttt{"colors"} \\
 & & \ \#\ \mathit{setColor}\ \mathit{green} \\
 & & \ \#\ \mathit{setBGColor}\ 0.2
\end{array}
$$


Abbildung 0.4.1: Attributierung zur Farbwahl

## 0.5 Frames

$$\mathit{box}, \mathit{circle}, \mathit{oval} \quad :: \quad \mathit{Picture} \to \mathit{Picture}$$



```
newshapes    =        -- shape drum
                      -- —— shape (fuzzy 345 67)
                      -- —— shape (fuzzy 346 67)
                    shape (setDX 10 ∘ triangle)
                     ▥ shape (triangle)
                          ▥ cloud 345 67 (tex "joe")

shape        ::  (IsPicture a, HasBGColor a)
             ⇒   (Picture → a) → Picture
shape f      =   setTrueBoundingBox
                       (draw [ref C -- ref C + 60 ∗ dir a
                                | a ← [0, 10 .. 360]]
                             (f (tex "joe") # setBGColor 0.97))
```

erwähne automatische Größenanpassung Attributierung; dx Boxen fester Größe

## 0.6 Kombination von Bildern

Bisher haben wir nur die Möglichkeit kennengelert, atomare Bilder zu erzeugen und diese mit einem Rahmen zu versehen. Nun wollen wir zwei Bilder zu einem neuen kombinieren. Für die häufigsten Kombinationen, nämlich die Anordnung nebeneinander und übereinander gibt es zwei Operatoren.

```
( ◫ )        ::  Picture → Picture → Picture

( ⊟ )        ::  Picture → Picture → Picture
```

Meistens ist aber zwischen den Bildern ein kleiner Abstand gewünscht. Für diesen Fall können wir jetzt schon mit unseren bisherigen Sprachmitteln die entsprechenden Operatoren ableiten.

```
( ▥ )        ::  Picture → Picture → Picture
a ▥ b        =   a ◫ hspace 8 ◫ b

( ▤ )        ::  Picture → Picture → Picture
a ▤ b        =   a ⊟ vspace 8 ⊟ b
```

Es wurde bisher noch nicht erwähnt, in welchen Einheiten Werte angegeben werden. METAPOST verwendet als Grundeinheit PostScript Punkte, d.h. $1/72$ Inch. Dies ist auch in *functional* METAPOST so. *hspace* 8 definiert also einen horizontalen Abstand von $1/9$ Inch was ungefähr $2.82$ mm entspricht. Um Abstände in anderen Einheiten angeben zu können, gibt es vordefinierte Konstanten mit denen die Werte entsprechend zu multiplizieren sind.

```
mm, pt, bp, cm, pc, cc, inch  ::  Numeric
mm                            =   2.84528
pt                            =   1
bp                            =   1.00375
cm                            =   28.45276
pc                            =   12
cc                            =   12.84010
inch                          =   72.27
```

*hspace* $(2 * cm)$ gibt also einen Abstand von zwei Zentimetern an.

*bsp30* = *tree1*
◫ *hspace* 10
◫ (`"$\\Longrightarrow$"` ⊟ `"right␣rotation"`)
◫ *hspace* 10
◫ *tree2*

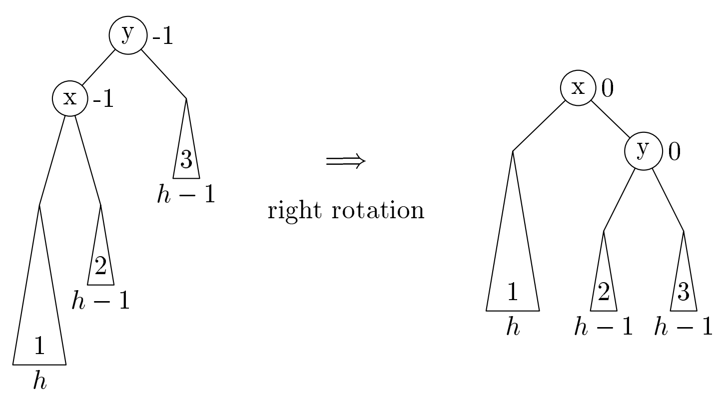



Abbildung 0.6.1: Plazierungen

```
bsp40              = box2 (tex "set␣of␣all␣languages"
                   ⊟   box2 (tex "rec.␣enum.␣languages␣or␣typ␣0"
                   ⊟   box2 (tex "decideable␣languages"
                   ⊟   box2 (tex "contextsensitive␣or␣typ␣1"
                   ⊟   box2 (tex "contextfree␣or␣typ␣2"
                   ⊟   box2 "regular␣or␣typ␣3--languages")))))
         where
         box2 a    = rbox 20 a
                   # setDX 8
                   # setDY 8
```

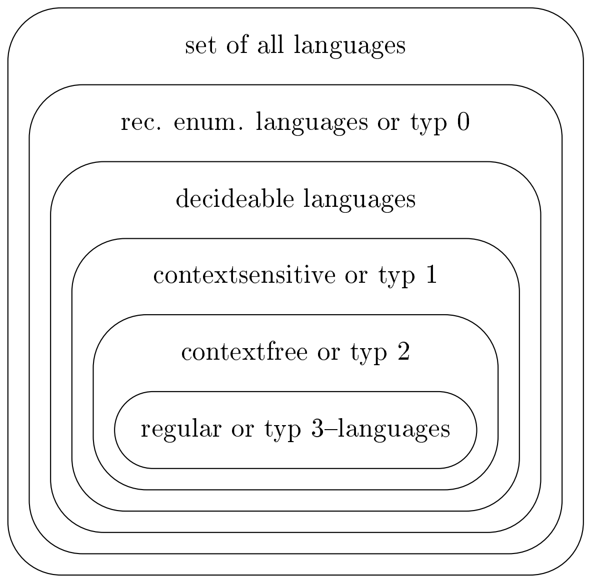



Abbildung 0.6.2: Kombination aus Rahmen und Plazierungen

Für mehr als zwei Bilder gibt es Funktionen, die eine Anordnung in einer Reihe bzw. Spalte bewirken.

$row :: [Picture] \to Picture$

$column :: [Picture] \to Picture$

Abbildung 0.6.3 zeigt noch eine Variante, bei der zwischen den Bildern ein zusätzlicher Abstand angegeben werden kann.

$rowSepBy :: Numeric \to [Picture] \to Picture$

$columnSepBy :: Numeric \to [Picture] \to Picture$

$$
\begin{aligned}
\mathit{bsp60} \quad = \quad & \mathit{box}\ (\mathit{columnSepBy}\ 10 \\
& \quad [\mathit{circle}\ \texttt{"R"} \\
& \quad\ \#\ \mathit{setDX}\ 10 \\
& \quad\ \#\ \mathit{setBGColor}\ \mathit{red}, \\
& \quad \mathit{circle}\ \texttt{"O"} \\
& \quad\ \#\ \mathit{setDX}\ 10 \\
& \quad\ \#\ \mathit{setBGColor}\ \mathit{yellow}, \\
& \quad \mathit{circle}\ \texttt{"G"} \\
& \quad\ \#\ \mathit{setDX}\ 10 \\
& \quad\ \#\ \mathit{setBGColor}\ \mathit{green}]) \\
& \#\ \mathit{setBGColor}\ \mathit{black} \\
& \#\ \mathit{setDX}\ 10 \\
& \#\ \mathit{setDY}\ 10
\end{aligned}
$$

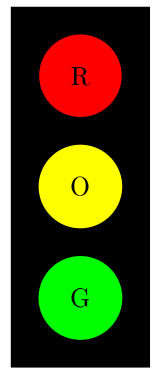



Abbildung 0.6.3: Das Bild einer Ampel (Siehe auch [**?**, FinneJones95]

In manchen Fällen reichen die vorgestellten Operatoren zur Anordnung von Bildern nicht aus. Abbildung 0.6.4 zeigt eine Situation, in der die Buchstabenreihen a und b bzw. c und d so untereinander angeordnet sind, daß der Buchstabe c unter b steht. Zur Lösung des Problems bedarf es einer Berücksichtigung der horizontalen und gleichzeitig der vertikalen Dimensionen. Dies berücksichtigt die Funktion

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

Die Funktion Matrix teilt die Ebene in Spalten und Reihen ein. Jede Spalte ist so brei, wie das breiteste Bild in ihr. Analog dazu ist jede Zeile so hoch, wie das höchste Bild in ihr. In den damit definierten rechteckigen Zellen wird jedes Bild zentriert plaziert.
Wie auch bei den Funktionen *row* und *column* gibt es eine Variante, bei der sich der ein zusätzlicher Abstand zwischen den Zellen horizontal wie vertikal angeben läßt.

$$\mathit{matrixSepBy} \quad :: \quad \mathit{Numeric} \rightarrow \mathit{Numeric} \rightarrow [[\mathit{Picture}]] \rightarrow \mathit{Picture}$$

$$
\begin{aligned}
\mathit{matrix1} \quad = \quad & \mathit{matrix}\ [[\mathit{tex}\ \texttt{"a"}, \mathit{tex}\ \texttt{"b"}], \\
& \quad [\mathit{empty}, \mathit{tex}\ \texttt{"c"}, \mathit{tex}\ \texttt{"d"}]]
\end{aligned}
$$

| | | |
|---|---|---|
| a | b | |
| | c | d |

Abbildung 0.6.4: Ausrichtung mit der Funktion *matrix*

## 0.7 Pfade

type Point,

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

$$\mathit{med} \quad :: \quad \mathit{Numeric} \rightarrow \mathit{Point} \rightarrow \mathit{Point} \rightarrow \mathit{Point}$$

type Numeric

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

hergeleitet:

$$
\begin{array}{lcl}
dir & :: & Numeric \to Point \\
dir\ a & = & vec\ (cos\ a, sin\ a) \\[1ex]
xy & :: & Point \to Point \to Point \\
xy\ p1\ p2 & = & vec\ (xpart\ p1, ypart\ p2) \\[1ex]
xdist & :: & Point \to Point \to Numeric \\
xdist\ p1\ p2 & = & xpart\ p1 - xpart\ p2 \\[1ex]
ydist & :: & Point \to Point \to Numeric \\
ydist\ p1\ p2 & = & ypart\ p1 - ypart\ p2
\end{array}
$$

Pfadsyntax

$$
\begin{array}{lcl}
(\ \text{--}\ ) & :: & (IsPath\ a, IsPath\ b) \Rightarrow a \to b \to Path \\
(\ ..\ ) & :: & (IsPath\ a, IsPath\ b) \Rightarrow a \to b \to Path \\
(\ \text{---}\ ) & :: & (IsPath\ a, IsPath\ b) \Rightarrow a \to b \to Path \\
(\ ...\ ) & :: & (IsPath\ a, IsPath\ b) \Rightarrow a \to b \to Path \\
(\ \&\ ) & :: & (IsPath\ a, IsPath\ b) \Rightarrow a \to b \to Path \\
cycle' & :: & Path
\end{array}
$$

$$
\begin{array}{lcl}
path1 & = & vec\ (20,20)\ \text{--}\ vec\ (0,0)\ \text{--}\ vec\ (0,30) \\
 & \text{--} & vec\ (30,0)\ \text{--}\ vec\ (0,0)
\end{array}
$$

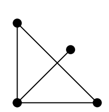


Abbildung 0.7.1: Ein einfacher Linienzug

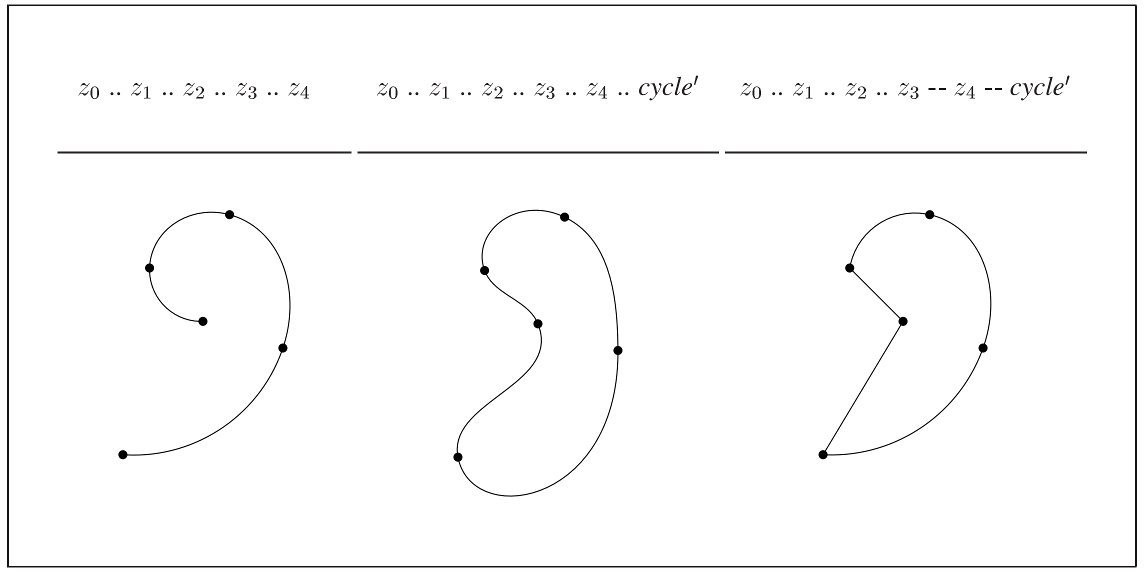
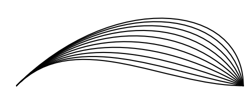



Abbildung 0.7.2: Die Wirkung verschiedener Pfadverbindunngen (Siehe figure 3 und 4 in [**?**])

$$\begin{aligned} \mathit{fig7} &= [\mathit{vec}\ (0,0) \mathinner{..} \mathit{vec}\ (3 * \mathit{cm}, 0) \\ &\qquad \#\ \mathit{setStartAngle}\ 45 \\ &\qquad \#\ \mathit{setEndAngle}\ (-10 * a) \\ &\quad \mid a \leftarrow [0\,..\,9]] \end{aligned}$$



$$\begin{aligned} \mathit{fig8} &= [\mathit{vec}\ (0,0) \mathinner{..} \mathit{vec}\ (3 * \mathit{cm}, 0) \\ &\quad\ \#\ \mathit{setStartAngle}\ 45 \\ &\quad\ \#\ \mathit{setEndAngle}\ (10 * a) \\ &\quad \mid a \leftarrow [0\,..\,7]] \end{aligned}$$

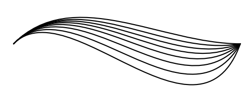


Abbildung 0.7.3: Start– und Endwinkel von Pfadsegmenten können vorgegeben werden (Siehe figure 7 und 8 in [**?**])

```
fig9 = rowSepBy 20 [(z0 .. z1
                       # setStartVector up
                       # setEndVector right) .. z2
                       # setEndVector down,
                      (z0 ... z1
                       # setStartVector up
                       # setEndVector right) ... z2
                       # setEndVector down]
```

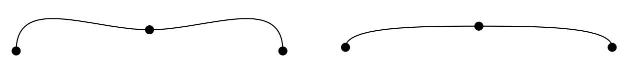


Abbildung 0.7.4: Der Unterschied zwischen den Pfadverbindungen ( .. ) und ( ... ) (Siehe figure 9 in [?]).

```
fig10 = rowSepBy 20 [
            z0 .. (z1 .. z2
                   # setJoin (joinTension (tension 1)))
             .. z3,
            z0 .. (z1 .. z2
                   # setJoin (joinTension (tension 2.3)))
             .. z3,
            z0 .. (z1 .. z2
                   # setJoin (joinTensions (tension 2.5)
                                          (tension 0.8)))
             .. z3]
```

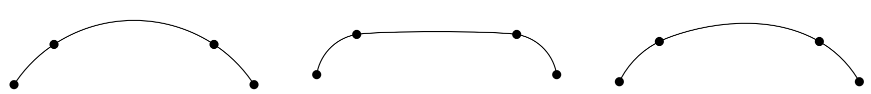


Abbildung 0.7.5: Der Parameter tension gibt den "Zug" an, der auf ein Pfadsegment wirkt (Siehe figure 10 in [?]).

```
draw :: [Path] → Picture → Picture
```

## 0.8 Strichmuster

```
dashPattern :: [Double] → Pattern
```

abgeleitet:

```
dashed    :: Pattern
dashed    =  dashPattern [3, 3]

dotted    :: Pattern
dotted    =  dashPattern [−1, 2.5, 0, 2.5]
```

## 0.9 Stifte

```
penSquare  :: (Numeric, Numeric) → Numeric → Pen

penCircle  :: (Numeric, Numeric) → Numeric → Pen
```

```
calligraphic  =  (transformPath (scaled 40) fullcircle
                      # setPen (penCircle (0.1, 5) 20))
              ▯▯    (transformPath (scaled 40) halfcircle
                      # setPen (penSquare (5, 5) 45))
              ▯▯    (transformPath (scaled 40) quartercircle
                      # setPen 5)
```

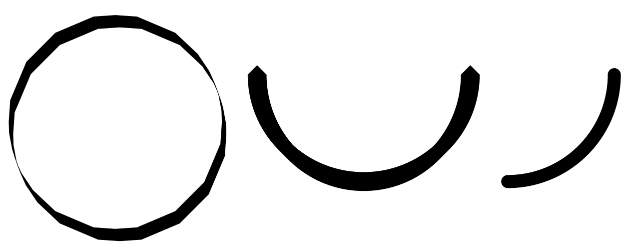


Abbildung 0.9.1: Kalligraphische Effekte

```
hBrack                    =  hBracket (vec (0,0)) (vec (100,0))
                 where
                 hBracket pl pr  =  pl
                       ... pl + vec (5,5)
                       --- med 0.5 pl pr + vec (−5,5)
                       ... med 0.5 pl pr + vec (0,10)
                       & med 0.5 pl pr + vec (0,10)
                       ... med 0.5 pl pr + vec (5,5)
                       --- pr + vec (−5,5)
                       ... pr
                          # setPen (penCircle (0.001,1) 0)
```

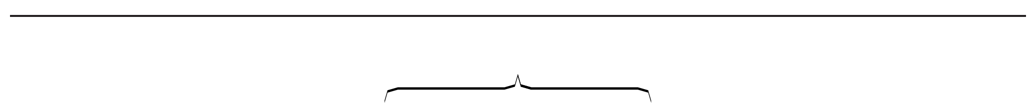


Abbildung 0.9.2: Eine horizontale Klammer

## 0.10 Pfeile

| | | |
|---|---|---|
| *defaultArrowHead* | :: | *PathArrowHead* |
| *arrowHeadSize* | :: | *Double* → *Double* → *PathArrowHead* |

abgeleitet:

| | | |
|---|---|---|
| *arrowHeadBig* | :: | *PathArrowHead* |
| *arrowHeadBig* | = | *pathArrowHeadSize* 8 4 |
| *default′* | = | *defaultArrowHead* |

verschiedene Pfeilarten

| | | |
|---|---|---|
| *ahFilled*, *ahLine* | :: | *ArrowHeadStyle* |
| *ahFilled* | = | *AHFilled* |
| *ahLine* | = | *AHLine* |

| | | |
|---|---|---|
| *setArrowHeadStyle* | :: | *ArrowHeadStyle* → *ArrowHead* → *ArrowHead* |
| *getArrowHeadStyle* | :: | *ArrowHead* → *ArrowHeadStyle* |

```
bsp50                      = pointerChain 25 ["12", "2", "2", "103"]
        where
        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 ∗ pt))
                            ◫   (box (toPicture bullet # setName "bullet")
                                  # setHeight (16 ∗ pt) # setWidth (16 ∗ pt))
```

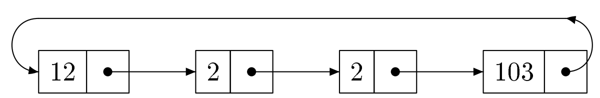



Abbildung 0.10.1: Darstellung einer verketteten Feldstruktur

## 0.11 Namen

**data** *Dir* $=$ *C* | *N* | *NE* | *E* | *SE* | *S* | *SW* | *W* | *NW*

Bezugspunkte: C,S,N,W,E.., ref

*setName* :: *Name* → *Picture* → *Picture*

(◁) :: *Name* → *Name* → *Name*
(⊲) :: *Int* → *Name* → *Name*

type Name + Operationen

## 0.12 Flächen

type Area

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

```
bsp4   = 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')
```

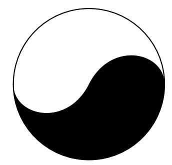


Abbildung 0.12.1: Beliebige zyklische Pfade lassen sich ausfüllen (Siehe figure 21 in [**?**]).

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

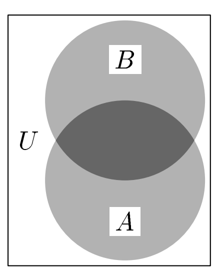



Abbildung 0.12.2: Ich komme ins RefManual (Siehe figure 22 in [**?**])

## 0.13 Clipping

| | | |
|---|---|---|
| *clip* | :: | *Path* → *Picture* → *Picture* |

## 0.14 Symbolische Gleichungen

| | | |
|---|---|---|
| *define* | :: | [*Equations*] → *Picture* → *Picture* |
| *define* | :: | [*Equations*] → *Path* → *Path* |
| *define* | :: | [*Equations*] → *Area* → *Area* |

type Equation

| | | |
|---|---|---|
| $(\doteq)$ | :: | $a \to a \to$ *Equation* |
| *equal* | :: | $[a] \to$ *Equation* |
| $(\dot{\equiv})$ | :: | $a \to a \to$ *Boolean* |
| $(\dot{\not\equiv})$ | :: | $a \to a \to$ *Boolean* |
| $(\dot{<})$ | :: | $a \to a \to$ *Boolean* |
| $(\dot{\leqslant})$ | :: | $a \to a \to$ *Boolean* |
| *whatever* | :: | *Point* |
| *whatever* | :: | *Numeric* |

```
bsp3            =   define [
        equal [ref "z1", -ref "z2", vec (0.2 * inch, 0)],
        equal [xpart (ref "z3"), -xpart (ref "z6"), 0.3 * inch],
        equal [xpart (ref "z3") + ypart (ref "z3"),
               xpart (ref "z6") + ypart (ref "z6"), 1.1 * inch],
        ref "z4"        ≐  med (1 / 3) (ref "z3") (ref "z6"),
        ref "z5"        ≐  med (2 / 3) (ref "z3") (ref "z6"),
        equal [ref "z20",
               med whatever (ref "z1") (ref "z3"),
               med whatever (ref "z2") (ref "z4")],
        equal [ref "z30",
               med whatever (ref "z1") (ref "z4"),
               med whatever (ref "z2") (ref "z5")],
        equal [ref "z40",
               med whatever (ref "z1") (ref "z5"),
               med whatever (ref "z2") (ref "z6")]]
                       (toPicture ["z1" -- "z20"
                                  -- "z2" -- "z30"
                                  -- "z1" -- "z40"
                                  -- "z2",
                                    "z1" -- "z2"
                                    # setPen 1,
                                    "z3" -- "z6"
                                    # setPen 1])
```

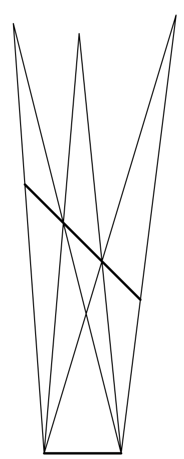
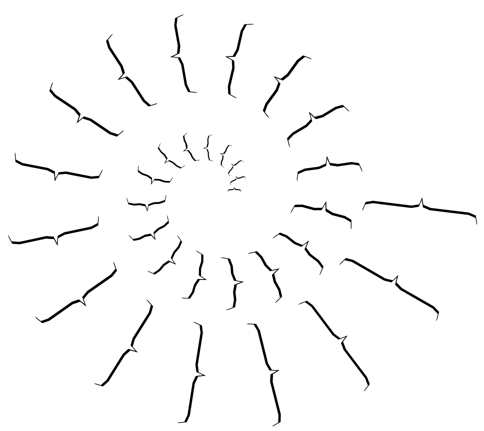


Abbildung 0.14.1: Unbekannte Werte werden hergelietet (Siehe figure 13 in [**?**]).

```
brack                    = [bracket ((5 + x / 14) ∗̇ dir x, (10 + x / 8) ∗̇ dir x)
                           | x ← [0, 23 .. 720]]

bracket                  :: (Point, Point) → Path
bracket (pl, pr)         = define [
        ref "start"      ≐ pl,
        ref "end"        ≐ pr,
        var "ang"        ≐ angle (ref "start" − ref "end"),
        var "d"          ≐ cond (dist (ref "start") (ref "end") <̇ 20)
                                (dist (ref "start") (ref "end") / 4)
                                5,
        ref "vecl"       ≐ var "d" ∗̇ dir (var "ang" − 135),
        ref "vecr"       ≐ var "d" ∗̇ dir (var "ang" − 45),
        ref "start2"     ≐ ref "start" + ref "vecl",
        ref "end2"       ≐ ref "end" + ref "vecr",
        ref "mid"        ≐ med 0.5 (ref "start") (ref "end")
                                 +(1.41 ∗ var "d")
                                 ∗̇ dir (var "ang" − 90),
        ref "midl"       ≐ ref "mid" − ref "vecl",
        ref "midr"       ≐ ref "mid" − ref "vecr"]
                 (pl ... ref "start2" --- ref "midl" ... ref "mid"
               & ref "mid" ... ref "midr" --- ref "end2" ... pr
                 # setPen (penCircle (0.001, var "d" / 5) (var "ang")))
```

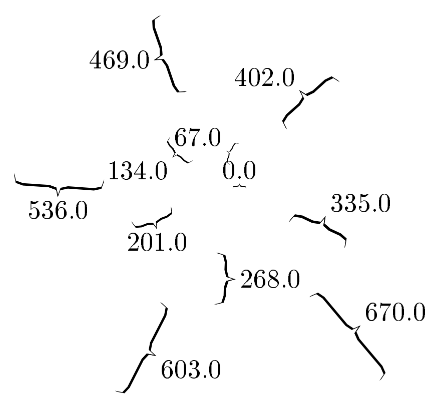


Beispiel 0.14.1: Klammern mit define

*overlay* :: [*Equations*] → [*Picture*] → *Picture*

```
brackLabel          = [bracket' x
                          ((5 + x / 14) * dir x, (10 + x / 8) * dir x)
                      | x ← [0, 67 .. 720]]

bracket'            :: IsPicture a ⇒ a → (Point, Point) → Path
bracket' l (pl, pr) = bracket (pl, pr)
                       # setLabel 0.5 C label
      where
      label         = overlay' [
            var "ang"   ≐ angle (pl − pr),
            ref (0◁C)   ≐ cond (var "ang" <̇ (−175.5)
                                   +175.5 <̇ var "ang")
                                       (ref (1◁S))
               (cond (var "ang" <̇ (−112.5)) (ref (1◁SE))
                (cond (    var "ang" <̇ (−67.5)) (ref (1◁E))
                 (cond (var "ang" <̇ (−22.5)) (ref (1◁NE))
                  (cond (var "ang" <̇ 22.5) (ref (1◁N))
                   (cond (var "ang" <̇ 67.5) (ref (1◁NW))
                    (cond (var "ang" <̇ 112.5) (ref (1◁W))
                          (ref (1◁SW))
                    )     )))))] (Just 0)
                             [empty, toPicture l]
```




Beispiel 0.14.2: Klammern mit “intelligentem“ Label

## 0.15 Transformationen

```
scale   :: Double → Picture → Picture
rotate  :: Double → Picture → Picture
```

```
fig28          = rek 4 empty
     where
     rek 0 pic = pic
     rek n pic = ooalign [draw [p] [toArea a # setColor 0.6,
                                    toArea p # setColor white],
                         rotate 90 (scale (1 / 3) (rek (n - 1) pic))]
     p         = transformPath (scaled 30) fullcircle
     a         = (vec (90, 0) .. vec (0, 30) .. vec (-90, 0)
                  # setEndCurl 1)
                  .. vec (0, -30) .. cycle'
                    # setEndCurl 1
```

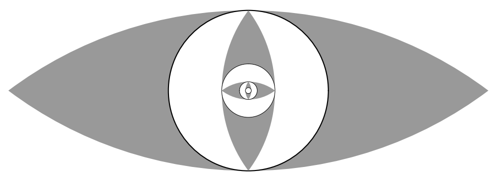


Abbildung 0.15.1: Eine rekursive Grafik (Siehe figure 28 in [**?**]).

$$
\begin{array}{l}
image\ bls = setTrueBoundingBox\,(\quad overlay'\ [\,]\ (Just\ 0)\\
\qquad\qquad\qquad\qquad\qquad ([BitLine\ (vec\ (0,y))\ Depth1\ bl\\
\qquad\qquad\qquad\qquad\qquad\quad |\quad (bl,y) \leftarrow zip\ bls\ [0,-0.12\,..]]))
\end{array}
$$

$$
\begin{array}{l}
cimage\ bls = setTrueBoundingBox\ (overlay'\ [\,]\ (Just\ 0)\\
\qquad\qquad\qquad\qquad\qquad ([BitLine\ (vec\ (0,y))\ Depth24\ bl\\
\qquad\qquad\qquad\qquad\qquad\quad |\quad (bl,y) \leftarrow zip\ bls\ [0,-0.30\,..]]))
\end{array}
$$

```
inverse [] = []
inverse (b : bs) = inv b : inverse bs
    where
    inv '0' = 'f'
    inv '1' = 'e'
    inv '2' = 'd'
    inv '3' = 'c'
    inv '4' = 'b'
    inv '5' = 'a'
    inv '6' = '9'
    inv '7' = '8'
    inv '8' = '7'
    inv '9' = '6'
    inv 'a' = '5'
    inv 'b' = '4'
    inv 'c' = '3'
    inv 'd' = '2'
    inv 'e' = '1'
    inv 'f' = '0'
    inv _ = '0'
```


Abbildung 0.15.2: Eine rekursive Grafik (Siehe figure 28 in [**?**]).