- Die erste Fassung von METAFONT wurde von Donald E. Knuth 1979 entwickelt, 1984 erschien eine revidierte, sehr viel elegantere Fassung.
- METAFONT erzeugt ein Raster mit einer einstellbaren Auflösung, das nur schwarze und weiße Pixel kennt.
- Das bedeutet, dass Farben nicht unterstützt werden und die Ausgabe von METAFONT nicht mehr skalierbar ist. Da Drucker METAFONT-Programme nicht ausführen können, muss im Vorfeld über die Auflösung entschieden werden.
- METAPOST wurde 1989-1994 von John D. Hobby entwickelt und bietet einen ähnlichen Sprachumfang wie METAFONT an, generiert aber PostScript.
- Das bedeutet, dass die Ausgabe skalierbar bleibt und direkt an einen PostScript-Drucker weitergereicht werden kann. Farben und Clipping werden unterstützt, rasterbezogene Operatoren fallen jedoch weg.

- Donald E. Knuth im METAFONTbook:

  The 'META-' part is more interesting: It indicates that we are interested in making high-level descriptions that transcend any of the individual fonts being described.
  [...]
  Meta-design is much more difficult than design; it's easier to draw something than to explain how to draw it. One of the problems is that different sets of potential specifications can't be easily be envisioned all at once. Another is that a computer has to be told absolutely everything. However, once we have successfully explained how to draw something in a sufficiently general manner, the same explanation will work for related shapes, in different circumstances; so the time spent in formulating a precise explanation turns out to be worth it.

- Der erste Namensteil META von METAFONT und METAPOST steht also dafür, dass eine Quelle eine ganze Familie von Schriftschnitten (oder Diagrammen) generieren kann.
- Im Unterschied zu PostScript sind METAFONT und METAPOST primär Quellformate für die Definition von Schriftschnitten und Zeichnungen.
- Viele PostScript-Schriftschnitte werden mit speziellen Werkzeugen entwickelt, die dann Type-1-Schriftschnitte generieren.
- Die Motivation von Donald E. Knuth für einen anderen Ansatz ergibt sich daraus, dass Schriftschnitte nicht mit akzeptablen Ergebnis auf triviale Weise skalierbar sind. Zahlreiche Parameter sind zu verändern, wenn besonders kleine oder große Schriftschnitte zu generieren sind.
- Bei PostScript gibt es eine ähnliche Möglichkeit nur über die sogenannten *hints* in den Type-1-Schriftschnitten.

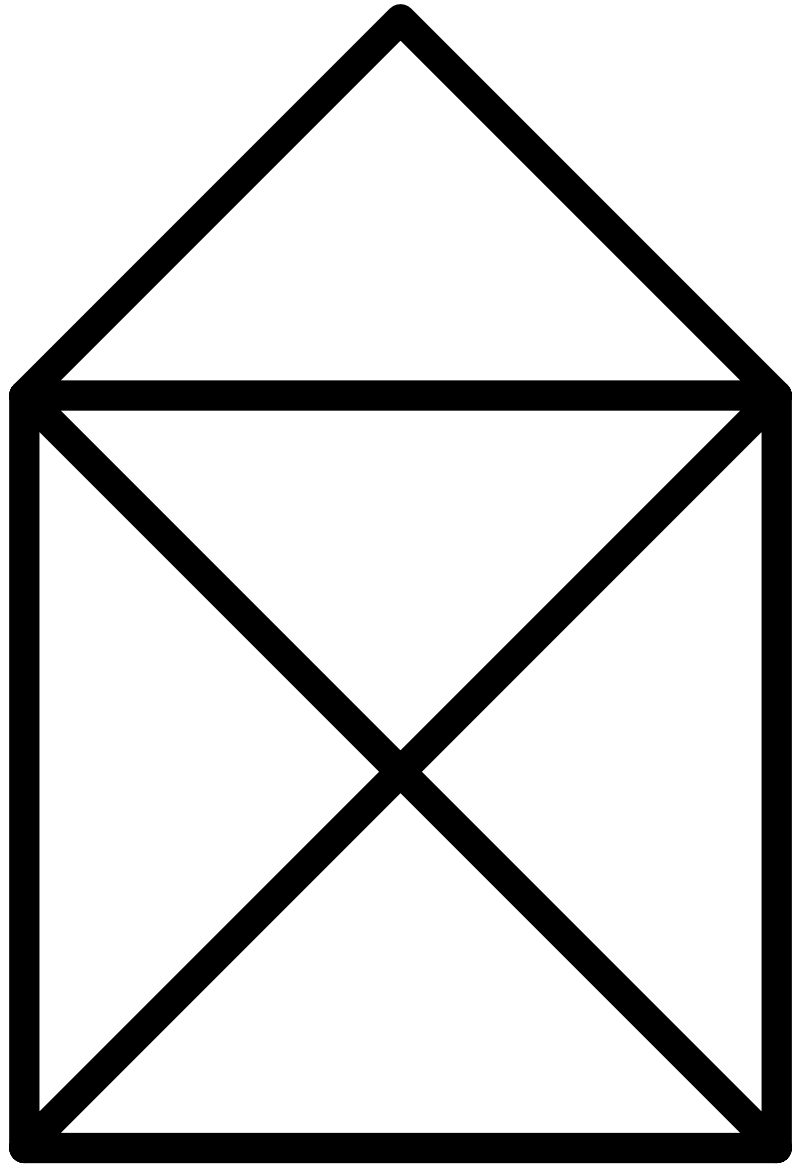
- Donald E. Knuth, *The METAFONTbook*, Addison-Wesley, 1986, ISBN 0-201-13445-4
- John D. Hobby, *A User's Manual for METAPOST*, `http://www.tug.org/docs/metapost/mpman.pdf`
- Alan Hoenig, *TEX Unbound*, Kapitel 13, *Using METAFONT and METAPOST*, 1998, ISBN 0-19-509686-X
- Michael Goossens et al, *The LATEX Graphics Companion*, Second Edition, Kapitel 3 und 4: *METAFONT and METAPOST: TEX's Mates*, *METAPOST Applications*, 2007, ISBN 0-321-50892-0

- METAFONT dient in erster Linie dazu, Schriftschnitte zu definieren.
- METAPOST unterstützt auch diesen Einsatzzweck (und erlaubte somit auch die skalierbare Einbettung von in METAFONT formulierten Schriftschnitten in PostScript).
- Im Gegensatz zu METAFONT unterstützt aber METAPOST die Verwendung von Schriften und ermöglicht damit die elegante Spezifikation von Diagrammen mit eingebetteten Texten.
- Als Vorbild diente hier auch das 1982 von Brian W. Kernighan entwickelte *pic*, das als Filter die Generierung von Diagrammen für *troff* unterstützte.
- Da METAPOST EPS-Dateien erzeugt, lässt es sich von jedem Text-System aus verwenden, das die Integration von EPS-Dateien ermöglicht.

house.mp

```
% Ein Haeusle mit 8 Strichen
prologues := 1;
beginfig(1);
   l := 1in; % Masseinheit fuer das Haeussle
   pickup pencircle scaled 3pt;
   draw (0,0) -- (0,1l) -- (0.5l,1.5l) -- (1l,1l) --
        (0,1l) -- (1l,0) -- (0,0) -- (1l,1l) -- (1l,0);
endfig;
end.
```

- Die Syntax erinnert an Pascal. Kommentare beginnen mit „%“.
- Die Zuweisung prologues := 1; sorgt dafür, dass die speziellen PostScript-Kommentare erzeugt werden, die im Einklang mit den PostScript-Strukturierungskonventionen stehen.
- Mit einem METAPOST-Programm lässt sich eine Vielzahl von EPS-Dateien erzeugen. `beginfig(1)` .. `endfig` umschließt das erste zu generierende Diagramm.
- Mit `end` wird das METAPOST-Programm beendet.



house.mp

```
l := 1in; % Masseinheit fuer das Haeussle
pickup pencircle scaled 3pt;
```

- Die Maßeinheiten entsprechen denen von PostScript. Es können aber auch diverse vordefinierte Maßeinheiten verwendet werden wie `pt` ($\frac{1}{72.27}$ Inch), *in* (Inch) oder `cm` (Zentimeter). Zusätzlich können (wie hier mit `l`) eigene Maßeinheiten definiert werden.
- Mit `pickup` wird ein neuer Zeichenstift definiert. Anders als in PostScript kann METAPOST beliebige geschlossene konvexe Pfade dafür nehmen. Hier liefert `pencircle scaled 3pt` einen kreisförmigen Zeichenstift mit einem Durchmesser von 3 Punkt. (Allerdings können nicht direkt Pfade an `pickup` übergeben werden, sie müssen noch zuvor mit der `makepen`-Funktion in einen Stift konvertiert werden. Bei `pencircle` ist dies bereits geschehen.)

house.mp

```
draw (0,0) -- (0,1l) -- (0.5l,1.5l) -- (1l,1l) --
     (0,1l) -- (1l,0) -- (0,0) -- (1l,1l) -- (1l,0);
```

- `draw` akzeptiert einen Pfad als Operanden und zeichnet diesen mit dem aktuellen Zeichenstift.
- Pfade können u.a. auch das Verbinden von Punkten mit geraden Strichen (unter Verwendung des Operators `--`) konstruiert werden.
- Die Idee eines aktuellen Pfades (wie bei PostScript) gibt es nicht. Stattdessen sind Pfade ganz normale Objekte, die in Variablen abgelegt werden können und auf die sich eine Reihe von Operatoren beziehen.

```
thales$ ls
house.mp
thales$ mpost house.mp
This is MetaPost, version 1.803 (kpathsea version 6.1.1)
(mpost.mp (/usr/local/texlive/2013/texmf-dist/metapost/base/plain.mp
Preloading the plain mem file, version 1.004) ) (./house.mp [1{psfonts.map}] )
1 output file written: house.1
Transcript written on house.log.
thales$ ls
house.1  house.log  house.mp
thales$
```

- Der METAPOST-Interpreter wird mit `mpost` aufgerufen.
- Dieser wird interaktiv, falls kein `end` in der Quelle vorkommt und nicht beim Aufruf von `mpost` mit der Option „-interaction batchmode“ auf Interaktionen verzichtet wurde.
- Für jede in `beginfig..endfig` eingeschlossene Zeichnung wird eine Ausgabedatei erzeugt, deren Namen mit dem Basisnamen der Quelle beginnt, gefolgt von einem Punkt und der Nummer der Zeichnung. Die Datei-Endung „.eps“ wird nicht angefügt.

```
thales$ mpost error.mp
This is MetaPost, version 1.803 (kpathsea version 6.1.1)
(mpost.mp (/usr/local/texlive/2013/texmf-dist/metapost/base/plain.mp
Preloading the plain mem file, version 1.004) ) (./error.mp
>> 0
! Improper ‘addto’.
<to be read again>
                   withpen
draw->...:also(EXPR0)else:doublepath(EXPR0)withpen
                                                .currentpen.fi._op_
<to be read again>
                   {
--->{
     curl1}..{curl1}
l.6    draw (0) --
                   (0,1l) -- (0.5l,1.5l) -- (1l,1l) --
? X
Transcript written on error.log.
thales$
```

- Wenn `(0,0)` innerhalb der Pfadkonstruktion durch `(0)` ersetzt wird, kommt es zu obigem Fehler.

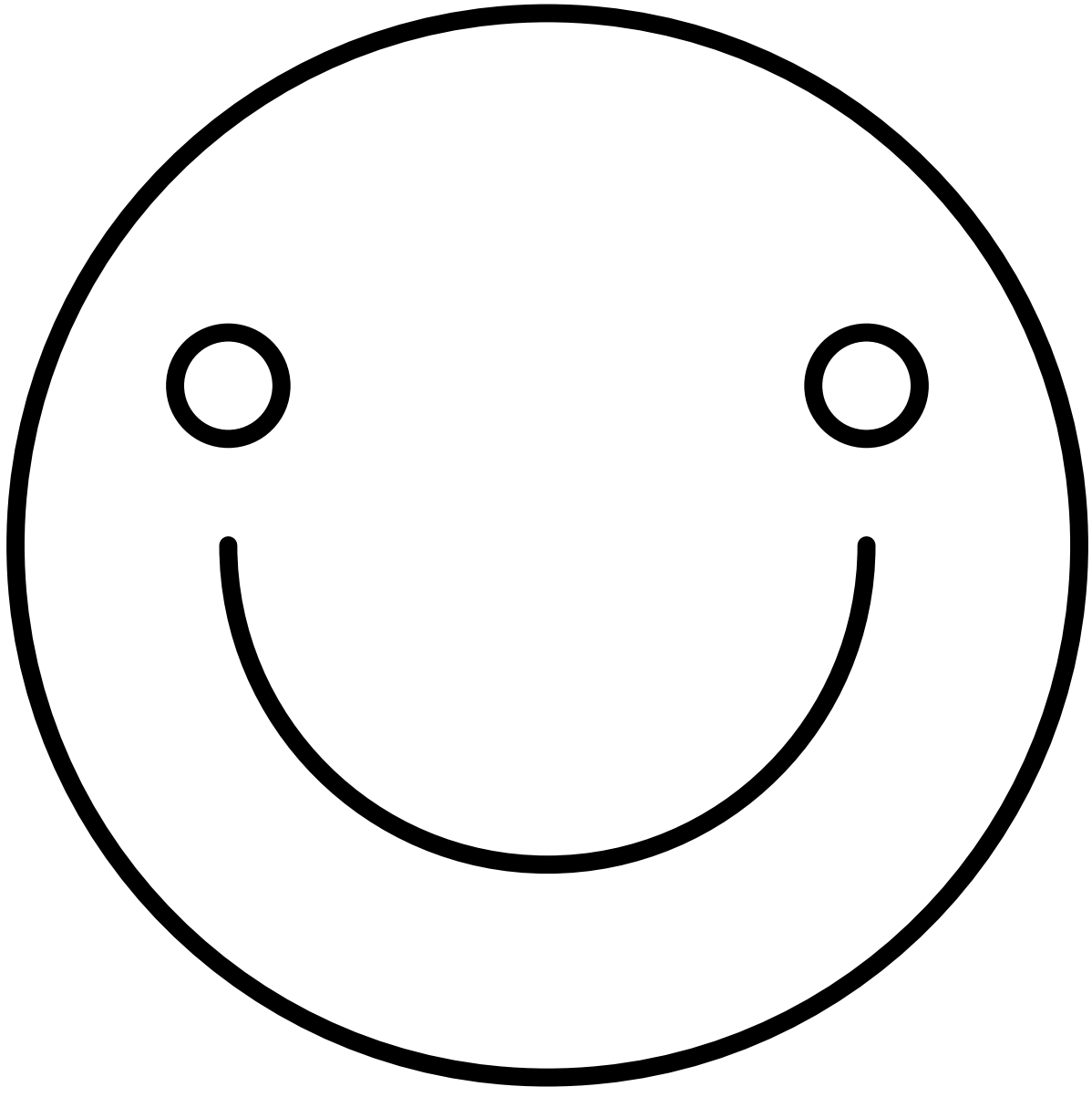
```
l.6    draw (0) --
                    (0,1l) -- (0.5l,1.5l) -- (1l,1l) --
? X
Transcript written on error.log.
thales$
```

- Die Fehlerstelle lässt sich ganz unten erkennen. „l.6“ steht für die Zeile 6 und die genaue Position lässt sich an dem Zeilenumbruch erkennen.
- METAPOST wird bei Fehlern interaktiv, es sei denn, dass dies von der Kommandozeile ausdrücklich ausgeschlossen wurde.
- Mit einer Eingabe von „X“ kann die METAPOST-Sitzung beendet werden. Ansonsten besteht die Möglichkeit, u.U. den Fehler interaktiv vorläufig zu korrigieren.

smiley.mp

```
% Ein Smiley
prologues := 1;
beginfig(1);
   l := 2in; % Radius des Gesichts
   pickup pencircle scaled 5pt;
   path circle;
   circle := (-1,0) .. (0,1) .. (1,0) .. (0,-1) .. cycle;
   draw circle scaled l; % Gesicht
   draw (-0.6l,0) .. (0,-0.6l) .. (0.6l,0); % Mund
   % linkes Auge
   draw circle scaled 0.1l shifted (-0.6l,0.3l);
   % rechtes Auge
   draw circle scaled 0.1l shifted (+0.6l,0.3l);
endfig;
end.
```

- Man beachte, dass sich `scaled 0.1l` nur auf den Pfad bezieht, nicht auf den Stift. (Bei PostScript ließ sich das nicht ohne weiteres trennen.)



smiley.mp

```
path circle;
```

- In METAFONT gibt es die 8 Datentypen boolean, string, path pen, picture, transform, pair und numeric.
- In METAPOST kommt noch der Datentyp color hinzu.
- Variablen, die nicht vom Typ numeric sind, müssen explizit mit ihrem Typnamen definiert werden.
- Variablennamen bestehen typischerweise aus Klein- und Großbuchstaben einschließlich dem Unterstrich _.
- Ziffern können nicht Bestandteil eines Variablennamens sein.
- Vordefinierte Namen dürfen nicht überdefiniert werden.
- Variablen-Deklarationen sind normalerweise global.

smiley.mp

```
path circle;
circle := (-1,0) .. (0,1) .. (1,0) .. (0,-1) .. cycle;
```

- In METAFONT/METAPOST können Pfade Variablen zugewiesen werden.
- Für Pfade gibt es zahlreiche Operatoren. In diesem Beispiel legt der Pfad-Operator `..` eine Bézier-Kurve zwischen einem Pfad und einen Punkt, so dass der Pfad ohne Ecken durch die beiden Punkte verläuft und vom Aussehen her optimiert wird.
- `cycle` ist ein spezieller Operator in einem Pfad-Ausdruck, der dem `closepath`-Operator in PostScript entspricht.

smiley.mp

```
draw circle scaled 1; % Gesicht
```

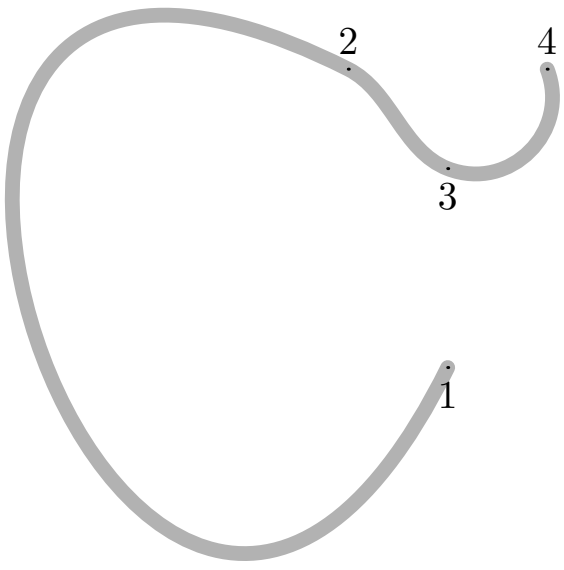
- Auf Pfade können diverse Transformations-Operatoren angewendet werden:

$$
\begin{aligned}
(x,y)\ \texttt{shifted}\ (a,b) &= (x+a, y+b)\\
(x,y)\ \texttt{scaled}\ s &= (sx, sy)\\
(x,y)\ \texttt{xscaled}\ s &= (sx, y)\\
(x,y)\ \texttt{yscaled}\ s &= (x, sy)\\
(x,y)\ \texttt{slanted}\ s &= (x+sy, y)\\
(x,y)\ \texttt{rotated}\ \theta &= (x\ \cos\theta - y\ \sin\theta, x\ \sin\theta + y\ \cos\theta)\\
(x,y)\ \texttt{zscaled}\ (u,v) &= (xu-yv, xv+yu)
\end{aligned}
$$

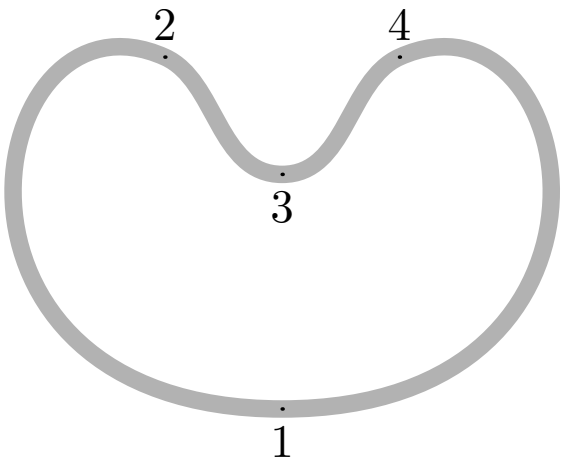
- Es gibt auch den generellen Operator `transformed`, der als rechten Operanden eine Variable oder einen Ausdruck vom Typ transform erwartet.

curves.mp

```
z1 = (0,-100); z2 = (-100,200);
z3 = (0,100); z4 = (100,200);
draw z1 .. z2 .. z3 .. z4;
```



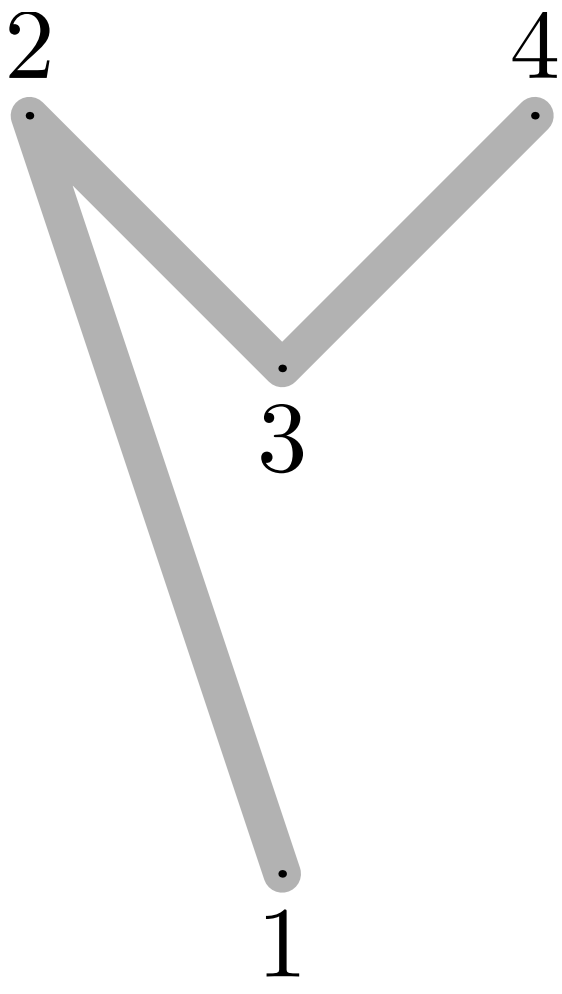
- z ist ein vordefiniertes Array von Punkten (Typ pair).
- z1 ist äquivalent zu z[1].
- Es gilt z1 = (x1,y1), d.h. x und y sind ebenfalls Arrays (vom Typ numeric), die mit dem Array z entsprechend als Aliase verknüpft sind.
- Der Operator .. konstruiert eine Bézier-Kurve, wobei darauf geachtet wird, dass aufeinanderfolgende Bézier-Kurven ohne Kanten ineinander übergehen.



curves.mp

```
z1 = (0,-100); z2 = (-100,200);
z3 = (0,100); z4 = (100,200);
draw z1 .. z2 .. z3 .. z4 .. cycle;
```

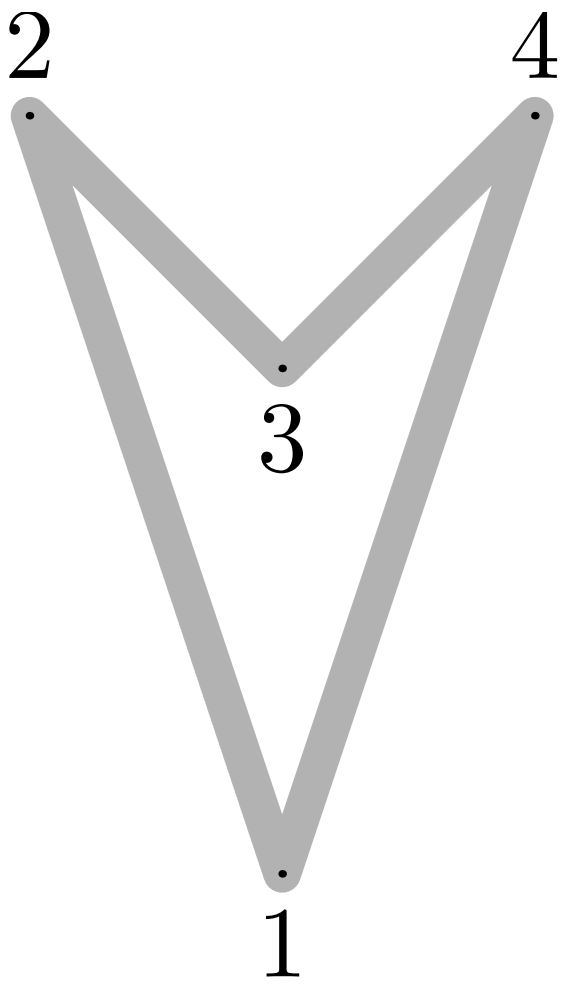
- Mit `cycle` wird der Pfad geschlossen.
- Das bedeutet in Kombination mit dem ..-Operator, dass die gesamte Kurve ohne Kanten gestaltet wird.



curves.mp

```
z1 = (0,-100); z2 = (-100,200);
z3 = (0,100); z4 = (100,200);
draw z1 -- z2 -- z3 -- z4;
```

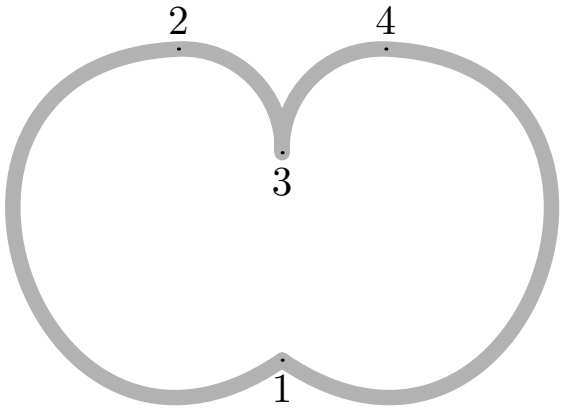
- Mit -- werden gerade Linien gezogen.



curves.mp

```
z1 = (0,-100); z2 = (-100,200);
z3 = (0,100); z4 = (100,200);
draw z1 -- z2 -- z3 -- z4 -- cycle;
```

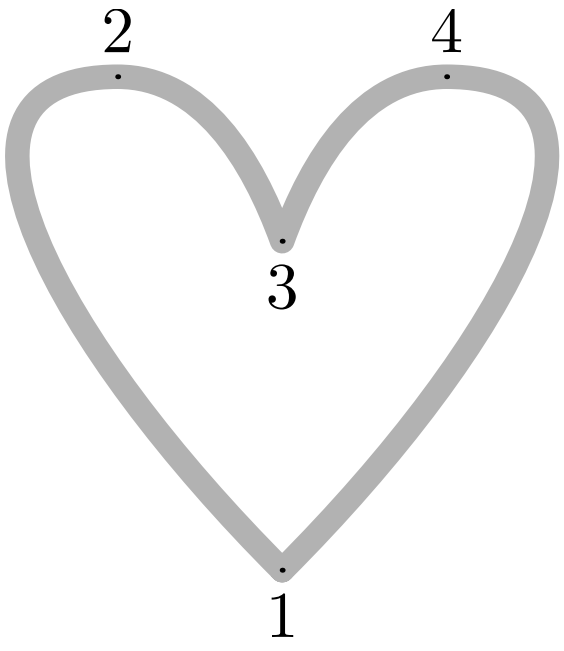
- Auch hier führt ein `cycle` dazu, dass die Kurve geschlossen wird.



curves.mp

```
z1 = (0,-100); z2 = (-100,200);
z3 = (0,100); z4 = (100,200);
draw z1 .. z2 .. z3 & z3 .. z4 .. z1;
```

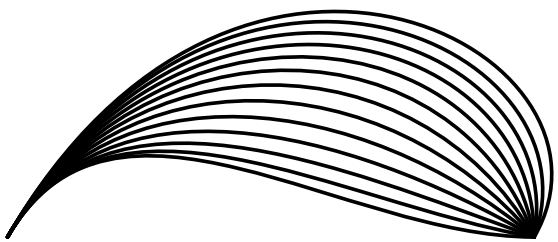
- Der Operator & verknüpft zwei Pfade miteinander, wobei keine Glättung der Kurve erzwungen wird.



curves.mp

```
z1 = (0,-100); z2 = (-100,200);
z3 = (0,100); z4 = (100,200);
draw z1{dir 135} .. z2{right} .. {dir -70}z3 &
     z3{dir 70} .. z4{right} .. z1{dir 225};
```

- Unmittelbar vor oder hinter einem Punkt kann eine Richtung angegeben werden, die dann jeweils diese Richtung in Richtung des Punktes bzw. ausgehend von dem Punkt erzwingt.
- Richtungen können als `left`, `right`, `up` and `down` spezifiziert werden. Alternativ kann die Richtung auch in Grad mit dem `dir`-Operator angegeben werden. Ferner ist es auch möglich, die Differenz zweier Punkte anzugeben.

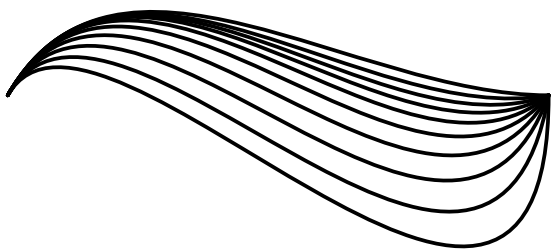


dcurves.mp

```
for d = 240 step 10 until 360:
   draw (0,0){dir 60} .. {dir d}(1in,0);
endfor
```

- Zu beachten ist hier, dass die unteren Kurven einen Wendepunkt haben, d.h. dass sie auf dem Wege von links nach rechts sich nicht immer nur nach rechts drehen, sondern ab einem Punkt in der Mitte nach links.
- Dieses und die folgenden zwei Diagramme wurden (in etwas modifizierter Form) dem METAFONTbook von Donald E. Knuth übernommen aus den Seiten 18 und 19.

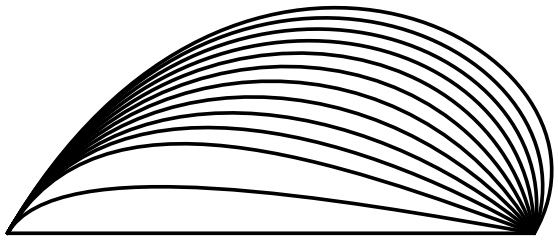
Angabe von Richtungen bei Pfaden



dcurves.mp

```
for d = 0 step 10 until 90:
   draw (0,0){dir 60} .. {dir d}(1in,0);
endfor
```

- Hier sind die Wendepunkte noch deutlicher zu sehen.



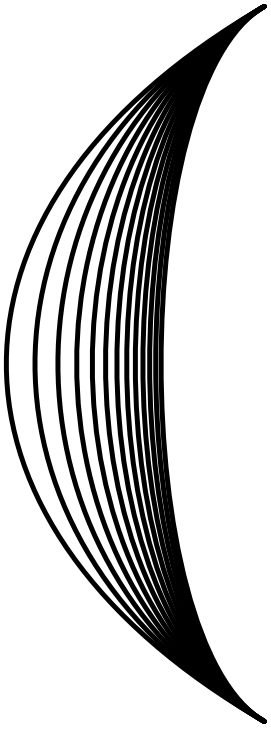
dcurves.mp

```
for d = 240 step 10 until 360:
   draw (0,0){dir 60} ... {dir d}(1in,0);
endfor
```

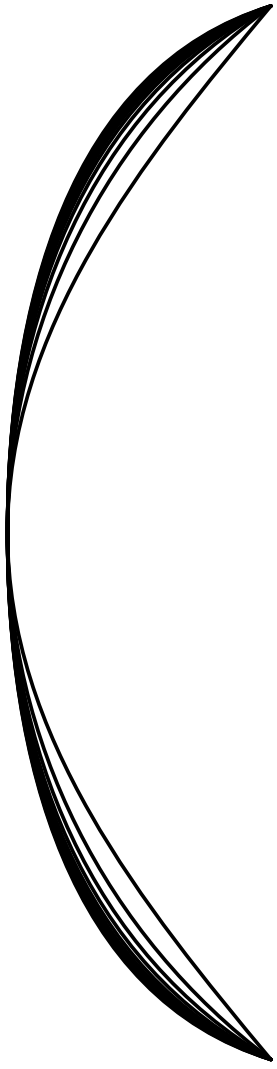
- Sei z_0 der linke Ausgangspunkt und z_1 der rechte Endpunkt.
- Dann gibt es möglicherweise einen Punkt z als Schnittpunkt der von z_0 und z_1 ausgehenden Strahlen entsprechend der angegebenen Richtungen.
- Falls es z gibt, dann liegt beim Operator „. . . “ die Kurve innerhalb des von z_0, z_1 und z gebildeten Dreiecks, d.h. ein Wendepunkt wird vermieden.
- Falls es kein z gibt, entspricht dieser Operator dem „. . “.

tension.mp

```
for t = -2 step 1 until 10:
   draw (0,0){dir 150} .. tension (1+t/10) .. {dir 30}(0,1in);
endfor
```



- Es ist möglich, die Spannung einer Kurve anzugeben. Per Voreinstellung wird eine `tension` von 1 gewählt.
- Je höher die Spannung wird, umso weiter nach rechts wird die Kurve in diesem Beispiel gedrückt.
- Die Spannung kann auch kleiner als 1 sein, muss aber mindestens den Wert $\frac{3}{4}$ haben. In diesen Fällen darf der Bogen sich noch weiter nach links ausstrecken.
- Das bedeutet, dass die Kontrollpunkte einer Bézier-Kurve umso näher zu den Ausgangs- und Endpunkten rücken, je höher die Spannung gewählt wird.



curl.mp

```
for c = 0 step 1 until 10:
   draw (0,0){curl c} .. (-0.5in,1in) .. {curl c}(0,2in);
endfor
```

- Wenn nichts anderes angegeben wird, entstehen annäherungsweise Kreisbögen.
- Mit `curl` kann an den Endpunkten ein Biegungsfaktor ausgewählt werden. Je höher dieser ist, umso steiler wird der Winkel, mit dem der Ausgangspunkt verlassen bzw. der Endpunkt erreicht wird.
- Ein Biegungsfaktor von 0 ist zulässig. In diesem Falle geht die Kurve fast direkt auf den Endpunkt zu. Per Voreinstellung liegt der Wert an den Endpunkten bei 1.
- Wenn `curl` inmitten einer Kurve angegeben wird, entsteht eine Knickstelle.

`ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz`	
`<=>:|`	
`‘’`	
`+-`	
`/*\`	
`!?`	
`#&@$`	
`^~`	
`[`	
`]`	
`{}`	
`.`	Bestandteil von Gleitkommazahlen; wird ansonsten ignoriert
`, ; ( )`	stehen jeweils für sich alleine
`"`	Beginn einer Zeichenkette
`0123456789`	Numerische Konstante
`%`	Kommentare (bis zum Ende der Zeile)

- METAFONT und METAPOST konvertieren den Programmtext in eine Sequenz von Symbolen.
- Leerzeichen, Zeilentrenner, Kommentare und Punkte trennen Symbole.
- Solange aufeinanderfolgende Zeichen der gleichen Zeichenklasse angehören, werden sie zu einem Symbol zusammengefasst.
- Eine besondere Behandlung erfahren numerische Konstanten, die mit einem Punkt (gefolgt von mindestens einer Ziffer) oder mit einer Ziffer beginnen. Es werden soviele Zeichen verschlungen, wie maximal zu einer numerischen Konstante gehören können.
- Zeichenketten gehen von `"` bis zum darauffolgenden `"`.
- Beispiele:

Eingabe	Symbole
`circle.radius`	„circle“, „radius“
`x2p`	„x“, „2“, „p“
`grmbl$!#@`	„grmbl“, „$“, „!“, „#@“

- In METAFONT und METAPOST werden alle eingelesenen Symbole, die keine Zeichenketten oder numerischen Konstanten sind, als Namen bezeichnet.
- Alle Namen fallen in eine von zwei Kategorien: Funken (*sparks*) und Etiketten (*tags*).
- Zu den Funken gehören alle vordefinierten Operatoren und die zusätzlich definierten Makros. Vordefinierte Variablen gehören **nicht** dazu.
- Beispiele für Funken:
  `beginfig draw path := + ( ) ; [`
- Beispiele für Etiketten:
  `c circle d l prologues x`

- Variablennamen bestehen aus einer Sequenz von Etiketten, bei denen auch numerische Konstanten enthalten sein können (jedoch nicht am Anfang).
- Mit einer Sequenz von Etiketten sind hierarchische Variablennamen möglich. Beispiele:
  `a` `a'` `a'b` `a'b::c`
- Wenn aufeinanderfolgende Etiketten der gleichen Zeichenklasse angehören, empfiehlt sich die Verwendung eines Punktes als Trenner. Beispiel: `a.b`
- Numerische Konstanten werden als Index eines Arrays interpretiert. Beispiel: `x2p` ist äquivalent zu `x[2].p` und
  `a1.5b` ist äquivalent zu `a[3/2]b`
- Die eckigen Klammern `[...]` sind nur notwendig, wenn für den Index ein Ausdruck verwendet wird.

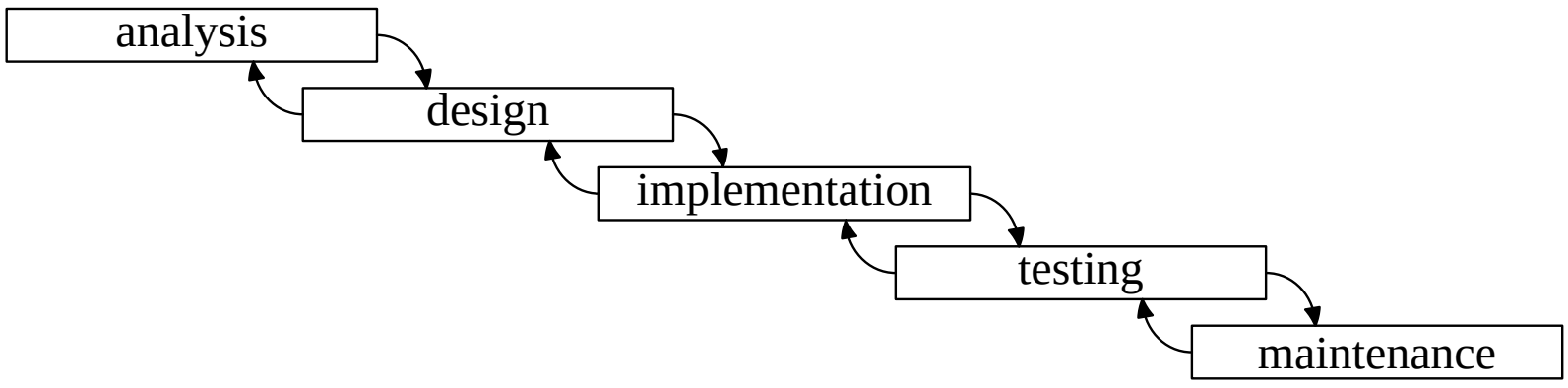
- Obwohl eine Notation wie `a.b` Verbundsstrukturen nahelegt, handelt es sich dabei um Hierarchien. Das bedeutet, dass *a* nicht für das Gesamtobjekt steht. Stattdessen kann `a` vom Datentyp `numeric` sein, während `a.s` vom Datentyp `string` ist und `a.p` den Datentyp `path` hat, wenn dies zuvor so deklariert wurde: `string a.s; path a.p`
- Es ist aber möglich, einen Teil eines hierarchischen Namens als Parameter bei einem Makro oder als zu durchlaufenden Wert bei einer `forsuffixes`-Schleife anzugeben. Entsprechend können hierarchische Namen genutzt werden, um verbundartige Strukturen zu repräsentieren.

- Neben dem Zuweisungsoperator := gibt es auch den Operator =, der nicht nur dem Vergleich dient, sondern auch die Spezifikation linearer Gleichungssysteme erlaubt.
- Beispiel:
  `a = 2b+c; 7b = c/2 - a; 4a = 5c+b - 3;`
  ist äquivalent zu:
  `a=1.92; b=-0.12; c=2.16;`
- Somit gibt es Variablen mit einem unbekannten Wert und ein System partiell aufgelöster Gleichungssysteme.

- Folgende Operationen mit unbekannten Werten sind zulässig:

  $-\ \langle \text{unknown} \rangle$
  $\langle \text{unknown} \rangle + \langle \text{unknown} \rangle$
  $\langle \text{unknown} \rangle - \langle \text{unknown} \rangle$
  $\langle \text{unknown} \rangle * \langle \text{known} \rangle$
  $\langle \text{known} \rangle * \langle \text{unknown} \rangle$
  $\langle \text{unknown} \rangle / \langle \text{known} \rangle$
  $\langle \text{known} \rangle\ [\langle \text{unknown} \rangle, \langle \text{unknown} \rangle]$
  $\langle \text{unknown} \rangle\ [\langle \text{known} \rangle, \langle \text{known} \rangle]$

Mit den vorgestellten Techniken lassen sich leicht Diagramme wie dieses Wasserfall-Modell erstellen:



- Die folgende Implementierung in METAPOST verfolgt folgende Ziele:
  - Es sollte leicht möglich sein, Phasen umzutaufen, hinzuzufügen oder wegzunehmen.
  - Alle Kästchen sollten genauso groß sein und den verwendeten Texten genügend Platz bieten.
  - Die Konfiguration, wie die Kästchen relativ zueinander platziert werden, sollte an einer zentralen Stelle erfolgen. Das gleiche gilt für die Anordnung der Pfeile.
- Wenn dies umgesetzt wird, kann nicht nur der Inhalt leicht variiert werden, sondern es können auch leicht verschiedene Parameter durchprobiert werden, um die gefälligste Variante zu ermitteln.

waterfall.mp

```
prologues := 1;
defaultfont := "ptmr8r"; % Times Roman
beginfig(1);
   % pictures containing the labels of the individual boxes
   picture stage.p[];
   % center, north/south/west/east point of the corresponding box
   pair stage.c[], stage.n[], stage.s[], stage.w[], stage.e[];
   string lbl; picture p;
   mw := 0; mh := 0; % maximal text width & height, seen so far
   stages := 0;
   % corners of our box
   x0 = x3 = -x1 = -x2 = maxwidth/2;
   y0 = y1 = -y2 = -y3 = maxheight/2;
   % configure all boxes
   % ...
   % construct our box
   maxwidth = mw * 5/4; maxheight = mh * 2;
   path box;
   box = z0 -- z1 -- z2 -- z3 -- cycle;
   % draw everything
   % ...
endfig;
end.
```

waterfall.mp

```
% corners of our box
x0 = x3 = -x1 = -x2 = maxwidth/2;
y0 = y1 = -y2 = -y3 = maxheight/2;
```

- Die Punkte z0, z1, z2 und z3 werden so definiert, dass sie ein Rechteck bilden.
- maxwidth und maxheight spezifizieren die Weite und die Höhe der Kästchen, sind aber zum Zeitpunkt dieser Gleichungen noch unbekannt. Sie werden erst dann bestimmt, wenn alle Beschriftungen ausgemessen worden sind.

waterfall.mp

```
% configure all boxes
for lbl = "analysis", "design", "implementation",
        "testing", "maintenance":
   stages := stages + 1;
   stage.p[stages] := p := thelabel(lbl, origin);
   % measure this label
   width := xpart(lrcorner p - llcorner p);
   if width > mw: mw := width; fi;
   height := ypart(urcorner p - lrcorner p);
   if height > mh: mh := height; fi;
   % position the corresponding box
   if stages = 1:
      stage.c[stages] = origin;
   else:
      stage.c[stages] - stage.c[stages-1] =
         (maxwidth*4/5, -maxheight*3/2);
   fi;
   % compute the connecting points of our box
   stage.n[stages] = 2/3[z0,z1] + stage.c[stages];
   stage.w[stages] = 1/2[z1,z2] + stage.c[stages];
   stage.e[stages] = 1/2[z0,z3] + stage.c[stages];
   stage.s[stages] = 2/3[z2,z3] + stage.c[stages];
endfor;
```

waterfall.mp

```
string lbl;
% ...
for lbl = "analysis", "design", "implementation",
          "testing", "maintenance":
   % ...
endfor;
```

- Die `for`-Schleife kann auch dazu verwendet werden, eine Reihe vorgegebener Werte zu durchlaufen.
- Nur an dieser Stelle wird entschieden, wieviele Kästchen gezeichnet werden, in welcher Reihenfolge sie erscheinen und welche Beschriftungen sie tragen.

waterfall.mp

```
    stage.p[stages] := p := thelabel(lbl, origin);
    % measure this label
    width := xpart(lrcorner p - llcorner p);
    if width > mw: mw := width; fi;
    height := ypart(urcorner p - lrcorner p);
    if height > mh: mh := height; fi;
```

- `thelabel` erzeugt ein Objekt des Datentyps `picture`, das eine Beschriftung enthält, die aus der angegebenen Zeichenkette in dem aktuellen Schriftschnitt (Variable `defaultfont`) erzeugt wurde.
- Der zweite Parameter spezifiziert die Position. Per Voreinstellung wird es zentriert. Aber es lässt sich auch relativ dazu positionieren, so würde `thelabel.bot` die Beschriftung unterhalb der angegebenen Position platzieren.
- Die Operatoren `lrcorner`, `llcorner`, `urcorner` und `lrcorner` liefern die Eckpunkte der Bounding-Box eines Pfades oder einer Zeichnung (Datentyp `picture`).
- Die Operatoren `xpart` und `ypart` selektieren die erste bzw. zweite Komponente eines Objekts mit dem Datentyp `pair`.

waterfall.mp

```
    % position the corresponding box
    if stages = 1:
       stage.c[stages] = origin;
    else:
       stage.c[stages] - stage.c[stages-1] =
          (maxwidth*4/5, -maxheight*3/2);
    fi;
```

- Das erste Kästchen wird um den Ursprung zentriert.
- Alle weiteren Kästchen werden jeweils relativ zum vorangegangenen Kästchen positioniert.
- Das vergrößert jeweils das Gleichungssystem, da zu diesem Zeitpunkt `maxwidth` und `maxheight` noch unbekannt sind, genauso wie alle Positionen mit Ausnahme der des ersten Kästchens.

waterfall.mp

```
% compute the connecting points of our box
stage.n[stages] = 2/3[z0,z1] + stage.c[stages];
stage.w[stages] = 1/2[z1,z2] + stage.c[stages];
stage.e[stages] = 1/2[z0,z3] + stage.c[stages];
stage.s[stages] = 2/3[z2,z3] + stage.c[stages];
```

- Der Operator $t\,[a, b]$ ist eine praktische Kurzform für $a + (b - a) * t$, die sowohl für numerische Werte als auch Objekte des Datentyps `pair` zulässig ist.
- Dieser Operator ermöglicht die elegante Spezifikation von Zwischenpunkten.
- Hier werden die Punkte festgelegt, von denen die Pfeile ausgehen und hinzeigen.
- Auch dies bereichert das Gleichungssystem, da sowohl die Punkte `z0` bis `z3` noch unbekannt sind wie auch `stage.c[stages]`.

waterfall.mp

```
maxwidth = mw * 5/4; maxheight = mh * 2;
```

- Erst durch diese beiden Gleichungen wird das gesamte Gleichungssystem lösbar.
- Erst wenn die Punkte alle bekannt sind, dürfen sie in einer Pfadkonstruktion verwendet werden:

waterfall.mp

```
path box;
box = z0 -- z1 -- z2 -- z3 -- cycle;
```

waterfall.mp

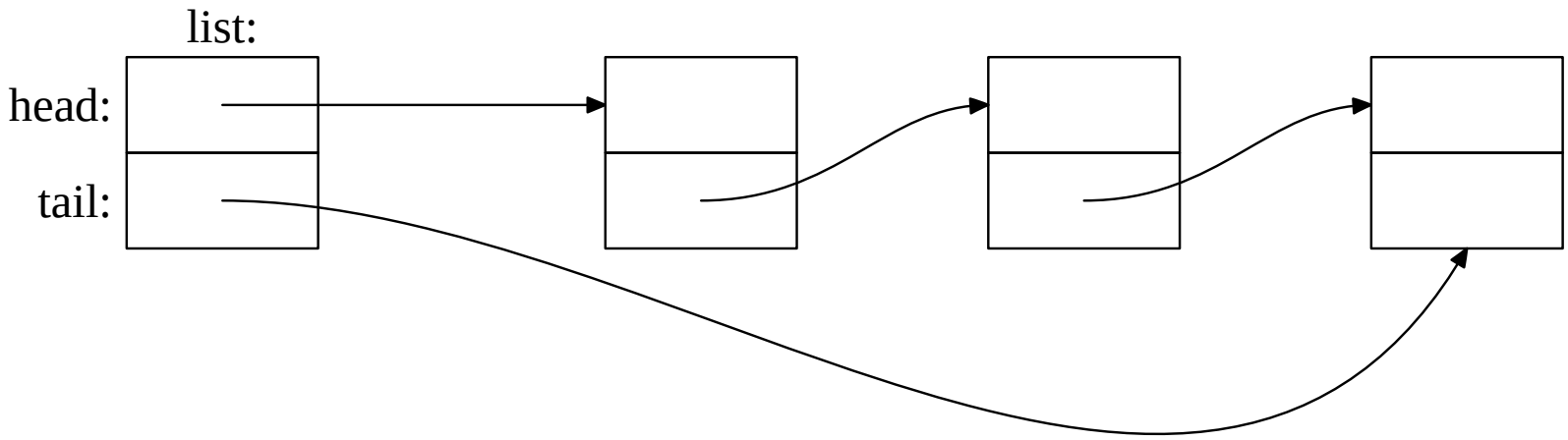
```
% draw everything
for i = 1 upto stages:
   draw box shifted stage.c[i];
   label(stage.p[i], stage.c[i]);
   if i > 1:
      drawarrow stage.e[i-1]{right} .. stage.n[i]{down};
      drawarrow stage.w[i]{left} .. stage.s[i-1]{up};
   fi;
endfor;
```

- `label` akzeptiert eine Zeichenkette oder ein bereits fertiges Objekt des Datentyps `picture` und zeichnet es an der angegebenen Position.
- `drawarrow` operiert wie `draw`, fügt aber am Ende noch einen Pfeil hinzu.

- Standardmäßig gehört zu METAPOST das boxes-Paket, das den Zusammenbau von Diagrammen mit Kästchen und Pfeilen entsprechend dem Vorbild von *pic* erlaubt.
- Die Idee liegt darin,
  - einzelne Kästchen (ggf. mit Beschriftungen) zu definieren,
  - die relativen Positionierungen der Kästchen zueinander mit Gleichungen zu beschreiben und
  - Pfeile, Linien, Beschriftungen und andere Figuren in Beziehung dazu zu setzen.

Beispielzeichnung mit `boxes`

Zeichnungen wie diese lassen sich leicht mit dem `boxes`-Paket spezifizieren:



list.mp

```
def node suffix $ =
   boxjoin(a.sw = b.nw; a.se = b.ne);
   forsuffixes $$ = $1, $2:
      boxit$$();
      ($$dx,$$dy) = (20pt,10pt);
   endfor;
enddef;
node list;
```

- METAFONT und METAPOST unterstützen Makrodefinitionen.
- Die Definition eines Makros besteht aus einem Namen (hier `node`), einer Parameterliste (hier `suffix $`) und einer Sequenz von Symbolen, die als Ersatztext dienen.
- Bei einer Makrodefinition wird der Ersatztext nur aufgesammelt. Er muss noch keiner Syntax genügen.
- Bei einem Makroaufruf wird eine Kopie des Ersatztextes mit ersetzten Parametern erzeugt. Diese Symbolsequenz wird dann an der Aufruf-Stelle eingefügt und erst dann parsiert.

- Bei Makros gibt es drei Parametertypen:

`expr`	beliebiger Ausdruck, der vor der Ersetzung ausgewertet wird
`suffix`	beliebiger Variablenname
`text`	beliebiger Text

- Der letzte Parameter kann (wie hier in diesem Beispiel) ohne Klammern spezifiziert werden. Alle vorangehenden Parameter benötigen Klammern und werden auch so spezifiziert.

Schleifen über Variablennamensteile

list.mp

```
forsuffixes $$ = $1, $2:
   boxit$$();
   ($$dx,$$dy) = (20pt,10pt);
endfor;
```

- `forsuffixes` setzt die Schleifenvariable auf die aufgezählten Namensbestandteile, die dann entsprechend innerhalb der Schleife jeweils ersetzt werden.
- In diesem Beispiel ist $$ die Schleifenvariable und $ der Makroparameter.
- Entsprechend stehen $1 und $2 für die Variable $ indiziert mit 1 und 2.

list.mp

```
boxjoin(a.sw = b.nw; a.se = b.ne);
```

- boxjoin ist ein Makro mit einem Text-Parameter:

```
def boxjoin(text equations) =
% ...
enddef
```

- Das Makro wird implizit beim Deklarieren einer Box aufgerufen mit jeweils a und b als Namen für die letzte und die gerade aktuelle Box.
- Der Aufruf findet aber nur statt, wenn eine letzte Box existiert und diese nach dem letzten boxjoin deklariert wurde.

Deklaration einer Box

list.mp

```
boxit$$();
```

- Boxen werden mit `boxit` deklariert. Eine Beschriftung kann über den Parameter spezifiziert werden (entweder als `string` oder als `picture`).
- Dabei handelt es sich um ein Makro mit einem Suffix-Parameter:

```
vardef boxit@# (text t) =
% ...
enddef
```

- `vardef` ist ähnlich wie `def`. Das Makro wird aber nur dort erkannt, wo Variablennamen zulässig sind.
- `@#` ist ein Spezialparameter bei `vardef`, einen beliebigen folgenden Namensbestandteil schluckt und über den Namen `@#` innerhalb des Makros zugänglich macht.
- Aufeinanderfolgende Boxen werden dann entsprechend den mit `boxjoin` deklarierten Gleichungen zusammengefügt.

list.mp

```
def draw_node (text $) =
   forsuffixes $$ = $:
      drawboxed($$1, $$2);
   endfor;
enddef;
```

- Ein `text`-Parameter kann insbesondere auch eingesetzt werden, um variable lange Listen von Variablennamen zu akzeptieren, die durch Kommata getrennt sind.
- Mit einer `forsuffixes`-Schleife ist es danach möglich, durch die Liste durchzuiterieren.
- `drawboxed` ist aus dem `boxes`-Paket und funktioniert nach dem gleichen Muster. Es zeichnet die angegebenen Boxen und positioniert sie, soweit dies nicht bereits durch vorangegangene Gleichungen festgelegt ist.

list.mp

```
node list;
members := 3;
for i = 1 upto members:
   node m[i];
   if i > 1:
      xpart m[i][1].c = xpart m[i-1][1].c + 80pt;
   else:
      xpart m[i][1].c = xpart list1.c + 100pt;
   fi;
   ypart m[i][1].c = ypart list1.c;
endfor;
draw_node(list
   for i = 1 upto members:
      , m[i]
   endfor);
```

- `for`-Schleifen generieren Textersatz, der an syntaktisch passender Stelle eingebettet werden kann.

list.mp

```
label.top("list:", list1.n);
label.lft("head:", list1.w);
label.lft("tail:", list2.w);
drawarrow list1.c -- m[1][1].w;
drawarrow list2.c{right} .. {dir 60}m[members][2].s;
for i = 2 upto members:
   drawarrow m[i-1][2].c{right}
      .. tension 3/4 and 1 .. {right}m[i][1].w;
endfor;
```

- Beschriftungen können mit dem Makro `label` angebracht werden.
- Wenn ein Etikett hinter `label` angegeben wird, legt es fest, in welcher Richtung relativ zum angegebenen Punkt die Beschriftung auszurichten ist. Fehlt sie, so wird die Beschriftung um den genannten Punkt zentriert.
- Folgende Ausrichtungen werden unterstützt:

`top`	überhalb des Punktes
`bot`	unterhalb des Punktes
`lft`	links neben dem Punkt
`rt`	rechts neben dem Punkt

- Bei strukturierten Diagrammen ist es sinnvoll, Makros für einzelne Bauelemente zu definieren, so dass nur noch Makroaufrufe notwendig sind, um den Bildinhalt zu definieren.
- Die Makros sollten dabei so funktionieren, dass sie eine geeignete Positionierung automatisch berechnen können — ggf. mit der Möglichkeit, dies überzudefinieren.

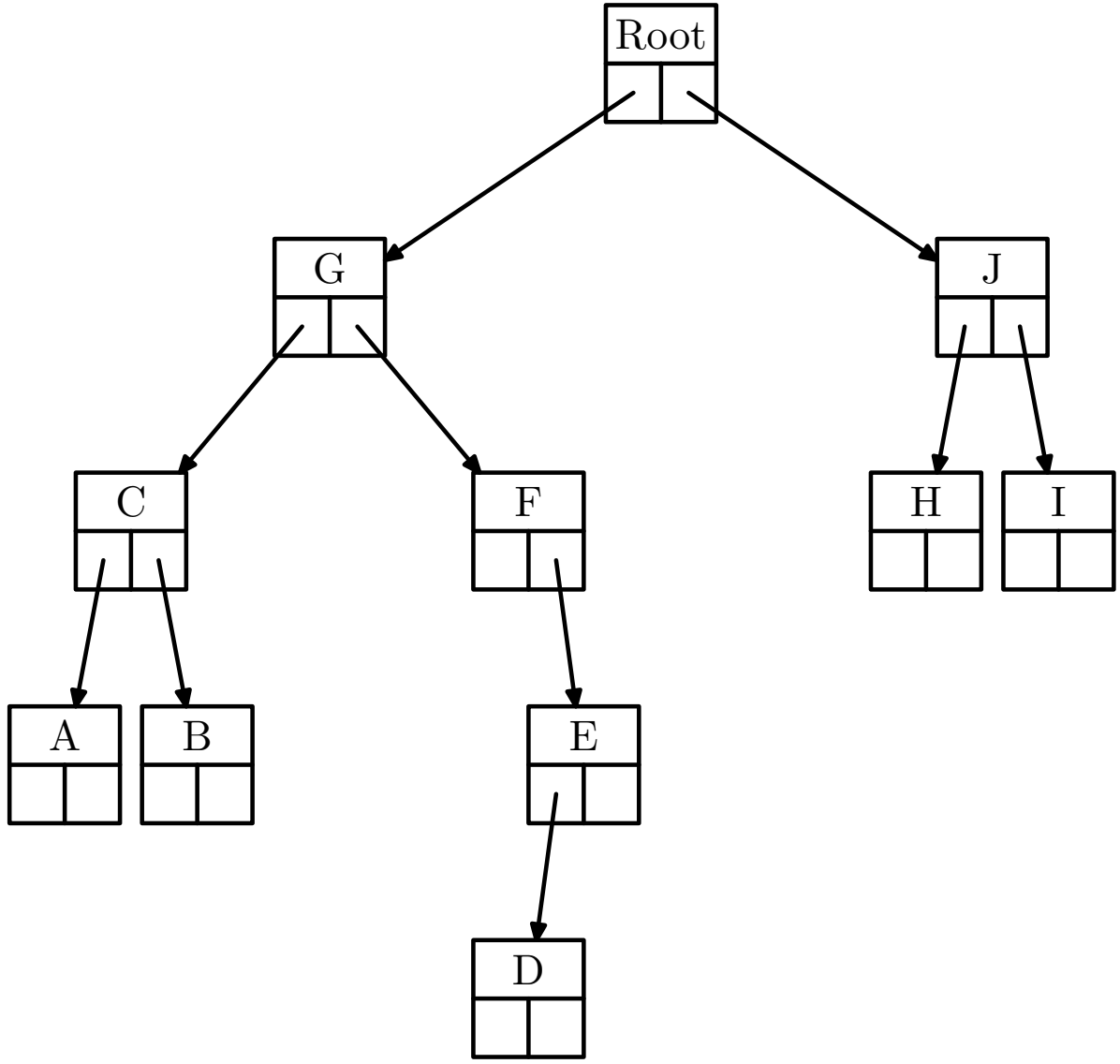
Die Makros sollten so strukturiert werden, dass

- sie beim Aufruf ein Bauelement mitsamt seinen Parametern aufnehmen, in die Datenstruktur aufnehmen und soweit möglich Gleichungen definieren und
- sie mit Hilfe von **vardef** zumindest eine Methode für das Zeichnen definieren, die nicht nur das eigene Bauelement zeichnet, sondern auch rekursiv alle benachbarten Bauelemente zeichnen lässt.

bintree.mp

```
beginfig(1);
   pickup pencircle scaled 1pt;
   Node(A, btex A etex, Nil, Nil);
   Node(B, btex B etex, Nil, Nil);
   Node(C, btex C etex, A, B);
   Node(D, btex D etex, Nil, Nil);
   Node(E, btex E etex, D, Nil);
   Node(F, btex F etex, Nil, E);
   Node(G, btex G etex, C, F);
   Node(H, btex H etex, Nil, Nil);
   Node(I, btex I etex, Nil, Nil);
   Node(J, btex J etex, H, I);
   Node(Root, btex Root etex, G, J);
   Root.Draw;
endfig;
```

- Die Idee ist, dass aus der Beschreibung eines binären Baums automatisiert eine geeignete grafische Repräsentierung gefunden wird, so dass sich keine Knoten versehentlich überlappen.



- So eine Lösung wäre akzeptabel.

- Jedes Makro, das als einen der Parameter einen **suffix** erhält, kann in diesem Namensraum mit **vardef** untergeordnete Makros definieren. Diese können wie objekt-orientierte Methoden später verwendet werden.
- Alle Parameter des übergeordneten Makros stehen auch dem untergeordneten Makro zur Verfügung analog zur Sprachtechnik der *closure* in Lisp, Scheme oder Perl. (Es handelt sich implementierungstechnisch natürlich nur um Textersatz, d.h. das untergeordnete Makro wird bei jedem Aufruf des übergeordneten Makros neu erzeugt und benötigt entsprechend weiteren Speicherplatz. Da die Makroparameter selbst nicht änderbar sind, führt dies zu keinem sichtbaren Unterschied.)
- In diesem Beispiel haben Objekte die Methode *Draw* zum Zeichnen, *Width* zum Ausmessen der benötigten Weite (damit es zu keinen Überlappungen kommt) und das **boolean**-Feld *node*, das bei Knoten wahr ist und bei Nil-Objekten unwahr.

bintree.mp

```
def Node(suffix $)(expr nodelabel)(suffix leftnode, rightnode) =
   boolean $.node; $.node := true;
   % ...
   vardef $.Width =
      % ...
   enddef;
   vardef $.Draw =
      % ...
   enddef;
enddef;

boolean Nil.node;
Nil.node := false;
vardef Nil.Width = 0 enddef;
vardef Nil.Draw = enddef;
```

- Zu beachten ist hier, dass *Nil* ein singuläres Objekt ist, während es beliebige viele Inkarnationen von *Node* geben kann, alle mit unterschiedlichen Namensraumpräfixen.

Es gibt zwei prinzipielle Ansätze, alles in Zusammenhang zu bringen und zu zeichnen:

- Alle Bauelemente sind hierarchisch organisiert. Entsprechend werden beim Makroaufruf für ein übergeordnetes Bauelement explizit alle untergeordneten Bauelemente genannt. Die *Draw*-Methode muss dann rekursiv auch die untergeordneten Bauelemente zeichnen lassen.
- Die Bauelemente sind unabhängig voneinander. Zusätzlich kommen Verbindungselemente (z.B. Pfeile) dazu, die jeweils die zu verbindenden Bauelemente gegeneinander positionieren und zeichnen lassen.

Im Falle der binären Bäume ist der erste Ansatz sinnvoll. Entsprechend gibt es die Parameter *leftnode* und *rightnode* für die beiden untergeordneten Knoten, wobei jeweils auch die Angabe von *Nil* zulässig ist.

Wenn (wie in diesem Beispiel) alle Knoten gleich groß und gleichzeitig für alle Beschriftungen groß genug gestaltet werden sollen, dann ist es notwendig,

- dafür Variablen zu verwenden, die in den Gleichungssystemen aufgenommen werden, und
- gleichzeitig Variablen zu verwalten, die die jeweilige Minima, Maxima oder sonstigen Berechnungszustände nach Deklaration aller bisherigen Bauelemente repräsentieren.

Die entsprechenden Variablen aus den Gleichungssystemen können dann beim ersten Aufruf einer *Draw*-Methode oder durch ein speziell dafür geschaffenes Makro gesetzt werden.

bintree.mp

```
% maintain maximal height and width of node labels
nodemaxheight := 0;
nodemaxwidth := 0;
```

- Die Variablen *nodemaxheight* und *nodemaxwidth* verwalten die bisherigen Maxima für die Höhe und die Weite der Beschriftungen der Knoten.
- In die Gleichungssysteme gehen die Variablen *nodeheight* und *nodewidth* ein.

Wie wird alles ausgemessen?

bintree.mp

```
pair $.c, $.n, $.s, $.e, $.w, $.nw, $.ne, $.se, $.sw;
$.height = nodeheight; $.width = nodewidth;
xpart $.n = xpart $.c = xpart $.s;
ypart $.w = ypart $.c = ypart $.e;
$.c = 1/2[$.n,$.s] = 1/2[$.w,$.e];
ypart $.n - ypart $.s = $.height;
xpart $.e - xpart $.w = $.width;
ypart $.nw = ypart $.n = ypart $.ne;
ypart $.sw = ypart $.s = ypart $.se;
xpart $.nw = xpart $.w = xpart $.sw;
xpart $.ne = xpart $.e = xpart $.se;
```

- Innerhalb des *Node*-Makros verwenden die Gleichungen die zu Beginn noch unbekannten Gleichungsvariablen *nodeheight* und *nodewidth*.

Wie wird alles ausgemessen?

bintree.mp

```
% measure caption and update nodemaxheight and nodemaxwidth, if necessary
picture $.caption;
$.caption = thelabel(nodelabel, origin);
cw := xpart(lrcorner $.caption - llcorner $.caption);
ch := ypart(ulcorner $.caption - llcorner $.caption);
if cw > nodemaxwidth:
   nodemaxwidth := cw;
fi;
if ch > nodemaxheight:
   nodemaxheight := ch;
fi;
```

- Innerhalb des Makros *Node* wird die übergebene Beschriftung ausgemessen und den bisherigen Maxima verglichen.

bintree.mp

```
if not known nodeheight:
   nodeheight = 4 nodemaxheight;
fi;
if not known nodewidth:
   nodewidth = 1.2 nodemaxwidth;
fi;
if not known $.c:
   $.c = origin;
fi;
```

- Dann wird innerhalb der *Draw*-Methode festgestellt, ob die Variablen *nodeheight* und *nodewidth* aus den Gleichungssystemen bereits bekannt sind. Falls nein, werden sie in Abhängigkeit von den zuvor ausgerechneten Maxima bestimmt.

bintree.mp

```
vardef $.Width =
   1.2 $.width + leftnode.Width + rightnode.Width
enddef;
```

- Die *Width*-Methode liefert in einem rekursiven Textersatz den Ausdruck für die gesamte Weite eines Unterbaums.
- Die Rekursion endet bei *Nil*:

bintree.mp

```
vardef Nil.Width = 0 enddef;
```

bintree.mp

```
boolean $.drawn;
$.drawn := false;
vardef $.Draw =
   if not $.drawn:
      $.drawn := true;
      % ...
   fi;
enddef;
```

- Bei rekursiven Zeichenprozeduren kann es sinnvoll sein, sich gegen mehrfache Aufrufe zu schützen. In streng hierarchischen Fällen (wie diesem) könnte darauf auch verzichtet werden.

bintree.mp

```
path $.p;
$.p := $.nw -- $.ne -- $.se -- $.sw -- cycle;
draw $.p;
draw $.w -- $.e;
draw $.c -- $.s;
draw $.caption shifted 0.5[$.n,$.c];
```

- Es ist sinnvoll, den Pfad des zu zeichnenden Bauelements explizit in einer Variablen abzuspeichern. Das erlaubt danach die elegante Verwendung des Pfads in **intersectionpoint**, wenn es darum geht, Pfeile oder andere verbindende Elemente passend einzuzeichnen.

bintree.mp

```
forsuffixes $$ = leftnode, rightnode:
   if $$.node:
      ypart $$.c = ypart $.c - 2 nodeheight;
   fi;
endfor;
if leftnode.node and rightnode.node:
   xpart leftnode.c + 1/2 * (leftnode.Width + rightnode.Width) =
      xpart rightnode.c;
   xpart $.c = xpart 0.5[leftnode.c,rightnode.c];
elseif leftnode.node:
   xpart leftnode.c = xpart $.c - 1/2 nodewidth;
elseif rightnode.node:
   xpart rightnode.c = xpart $.c + 1/2 nodewidth;
fi;
leftnode.Draw; Arrow(0.5[$.c,$.sw], leftnode);
rightnode.Draw; Arrow(0.5[$.c,$.se], rightnode);
```

- Zunächst werden hier die untergeordneten Knoten passend vertikal platziert. Danach erfolgt über die *Width*-Methode eine geeignete horizontale Positionierung.

bintree.mp

```
def Arrow(expr pfrom)(suffix target) =
   if target.node:
      path dline;
      dline := pfrom -- target.c;
      pair pto;
      pto := dline intersectionpoint target.p;
      drawarrow pfrom -- pto;
   fi;
enddef;
```

- **intersectionpoint** bestimmt den Schnittpunkt zweier Pfade.
- Wenn zwei Pfade sich mehrfach kreuzen, wird der »früheste« Schnittpunkt bestimmt nach einer etwas speziellen Definition (siehe Seite 137 im METAFONT-Buch).