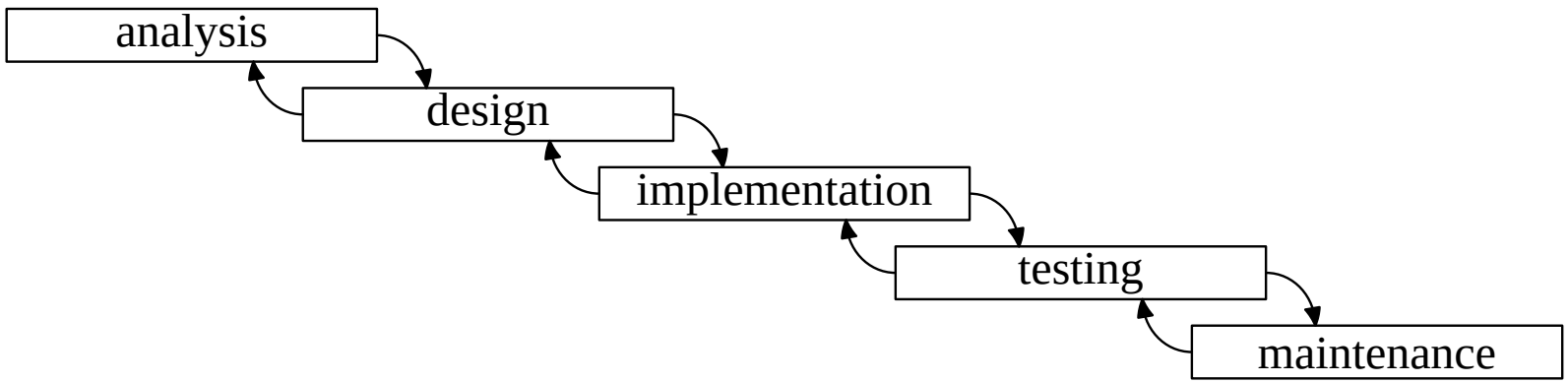
Mit den vorgestellten Techniken lassen sich leicht Diagramme wie dieses Wasserfall-Modell erstellen:



- 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

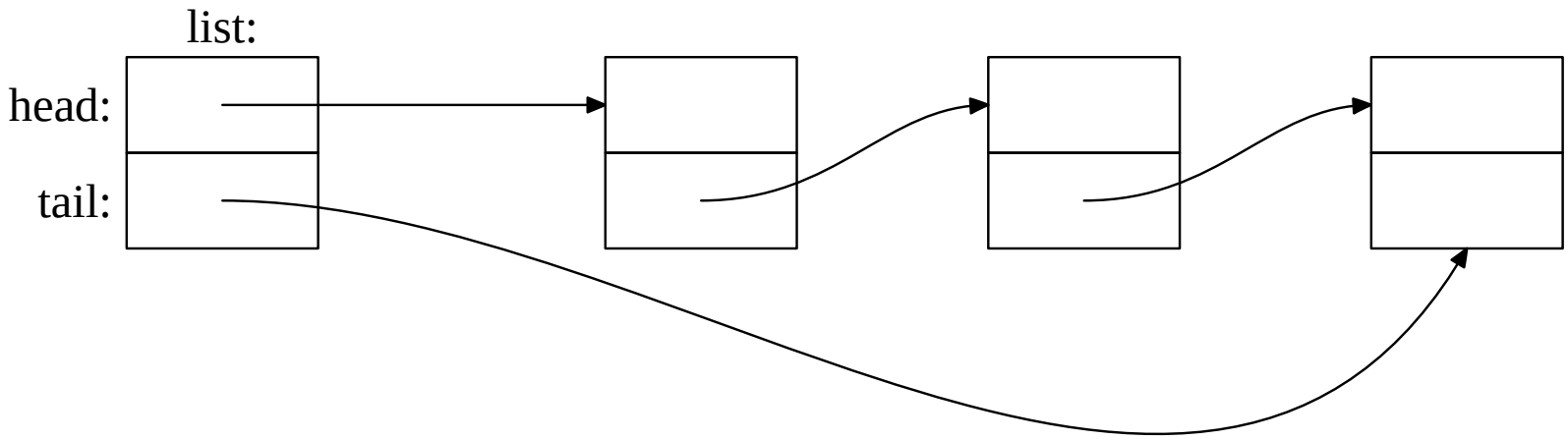
```
% draw everything
for i = 1 upto stages:
   draw box shifted stage.c[i];
   label(stage.p[i], stage.c[i]);
   if i > 1:
      drawarrow stage.e[i-1]{right} .. stage.n[i]{down};
      drawarrow stage.w[i]{left} .. stage.s[i-1]{up};
   fi;
endfor;
```

- *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.

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.

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

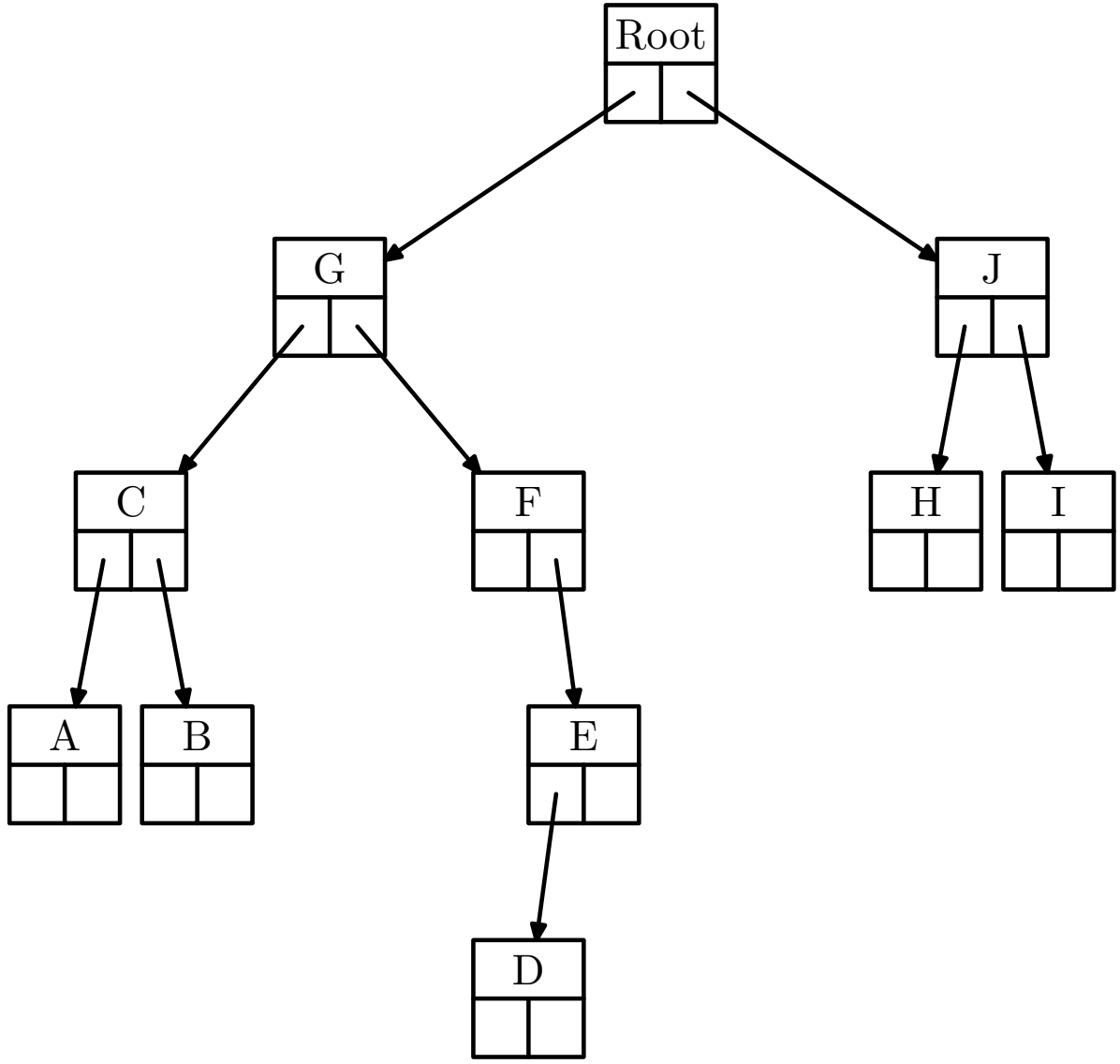
- sie beim Aufruf ein Bauelement mitsamt seinen Parametern aufnehmen, in die Datenstruktur aufnehmen und soweit möglich Gleichungen definieren und
- sie mit Hilfe von **vardef** zumindest eine Methode für das Zeichnen definieren, die nicht nur das eigene Bauelement zeichnet, sondern auch rekursiv alle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

Zeichnen von binären Bäumen



- 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.

Wie wird alles ausgemessen?

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).