

MetaObj:
Very High Level Objects
in MetaPost

Denis Roegel, University of Nancy, France

TUG 2002
Thiruvananthapuram, Kerala, India
4–7 September 2002

Summary

- The need for high-level objects
- How high-level objects can be implemented
- Examples
- Extensions

The need for objects

- need to separate *design* and *use* : allow for reusability and homogeneity

1. design/use intermeshed:

```
draw (0cm,0cm)--(1cm,0cm)--(1cm,1cm)
      --(0cm,1cm)--cycle;
```

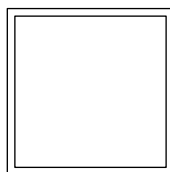
2. – design:

```
def square = (0cm,0cm)--... enddef
```

– use: draw square

- need to encapsulate complex structures so that they can be manipulated globally; drawing is only one of many possible manipulations;

problem with



if we write:



```
def doublesquare = ...  
draw doublesquare
```

it will only work if we make the double square a MetaPost picture; so, can we use pictures as objects?

- pictures vs. objects

- + pictures *are* structured objects;

- pictures can't be annotated and they contain only *visible* information, not other information that could influence the use of the picture;

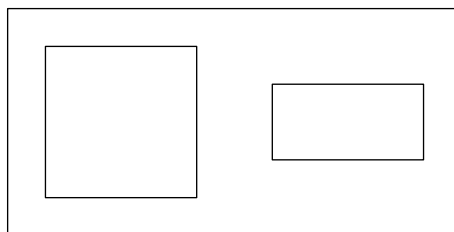
for instance, let  and  be two objects representing letters; if the objects contain only the drawing, doing *kerning* will be difficult; we

would have to find the properties of an object by guessing inside the picture, or by using information stored elsewhere;

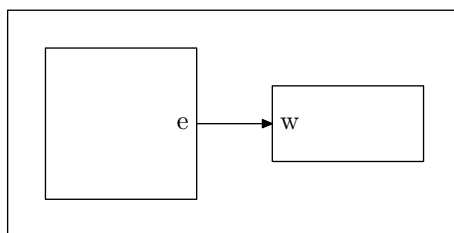
⇒ in a real object, we add margins, and all necessary properties to *use* and combine objects.

- need to go inside an object:

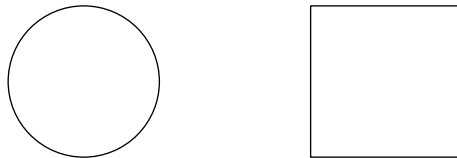
from



we would like to obtain



- when creating objects, we can also create different classes of objects, corresponding to different types of objects, with different uses and different attributes; example:



the circle has a *radius*, which is not a property of the square; also, the square has specific *corner positions*, which are not relevant in the circle;

- need for named subcomponents; the components of a picture do not have names;

All these things are cumbersome to handle without an object approach.

How objects are implemented: a look at `boxes.mp`

```
boxit.a(btex test etex);  
a.dx=a.dy;  
a.c=origin;  
drawboxed(a);
```



test

Interesting features:

- ★ boxes have names ('a');
- ★ boxes have parameters: dx, dy, c (and others), accessed in a natural dot-style way;
- ★ boxes can be “floating” (or “gliding”) before a.c is set;

- ★ if we dig more inside `boxes.mp`, we find that a box has several *points* (n, w, s, e, nw, ..., c) as well as *equations*;
- ★ two kinds of objects: boxes (created with `boxit`) and ellipses (created with `circleit`);
- ★ certain variables can be changed and they have default values;

We want to retain most of these features.

How objects are implemented: basic requirements

- objects will have *names*;
- objects will have properties (*attributes*);
- objects will have *equations*;
- when an object is created, it will be *rigid* and *floating* (not exactly true of `boxes.mp` where `dx` and `dy` need not be set when a box is created);
- we want *classes* of objects, but no inheritance so far;
- we want *class methods* as well as *object methods*;

- we want to apply certain *transformations* regardless of the object;
- objects must be *compatible* (and most objects should be interchangeable);
- compatibility with `boxes.mp`;
- objects must be able to contain objects;
- each class must have a *constructor*;
- the structure of an object must be easily accessible.

Floating objects

- floating point:

pair A;

⇒ A is floating;

- floating segment:

pair A,B;

$B-A=(1\text{cm},3\text{cm})$;

⇒ neither A nor B is defined, but the \overrightarrow{AB} vector is well defined.

The equations create the *rigidity*.

Operations on floating objects

- translations: not relevant;
- rotation: example:

```
pair A,B;  
B-A=(1cm,3cm);
```

```
pair C,D;  
C=A;D=B; % A and B are saved  
A:=(whatever,whatever); % re-initialize A  
B:=(whatever,whatever); % re-initialize B  
B-A=(D-C) rotated 30; % reset B-A
```

As a result, the segment has been rotated, and the segment is still floating.

Examples

A simple object: `EmptyBox`

`EmptyBox` is the constructor of a box containing nothing. Normally, it is invisible, but we can force its edges to appear.



This is constructed as follows:

```
newEmptyBox.eb(2cm,1cm) "framed(true)";
```

- `'newEmptyBox'` is the constructor for the `'EmptyBox'` class (similar to `boxit` in `boxes.mp`);
- `'eb'` is the name of the object (like with `boxes.mp`);

- the initial dimensions are given as parameters; these parameters are *mandatory*;
- an *option* is added to have the box framed; the option has a name and a value;

Once the box is constructed, it is floating and can be drawn with:

```
eb.c=origin;  
drawObj(eb);
```

Inside (!) the EmptyBox

```
vardef newEmptyBox@#(expr dx,dy) text options=  
  ExecuteOptions(@#)(options);  
  assignObj(@#,"EmptyBox");  
  StandardInterface;  
  ObjCode StandardEquations,  
    "@#ise-@#isw=(" & decimal dx & ",0)",  
    "@#ine-@#ise=(0," & decimal dy & ")";  
enddef;  
  
def BpathEmptyBox(suffix n)=  
  StandardBpath(n)  
enddef;  
  
def drawEmptyBox(suffix n)=  
  if show_empty_boxes:  
    drawFramedOrFilledObject_(n);  
  fi;  
enddef;  
  
setObjectDefaultOption  
  ("EmptyBox")("framed")(false);
```

Inside EmptyBox (1)

```
vardef newEmptyBox@#(expr dx,dy) text options=  
  ExecuteOptions(@#)(options);  
  assignObj(@#,"EmptyBox");  
  StandardInterface;  
  ObjCode StandardEquations,  
    "@#ise-@#isw=(" & decimal dx & ",0)",  
    "@#ine-@#ise=(0," & decimal dy & ")";  
enddef;
```

- '@#' is the name of the box;
- options may be empty;
- the values of the options for the current object are stored with `ExecuteOptions` (a little bit like in $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$);
- `assignObj` initializes various things, including the correspondence between an object and its class;
- `StandardInterface` and `ObjCode` will be explained later.

Inside EmptyBox (2)

```
def BpathEmptyBox(suffix n)=  
    StandardBpath(n)  
enddef;
```

```
def drawEmptyBox(suffix n)=  
    if show_empty_boxes:  
        drawFramedOrFilledObject_(n);  
    fi;  
enddef;
```

- all objects must define a path bounding the object; the name is mandatory, here, `BpathEmptyBox`; it is a *method*;
- all objects must defined a *drawing method*, named here `drawEmptyBox`.

Inside EmptyBox (3)

```
setObjectDefaultOption
```

```
("EmptyBox")("framed")(false);
```

- options are given to the constructor as a *comma-separated list* of strings;
- there are various named options;

Option	Type	Default
<i>filled</i>	boolean	false
<i>fillcolor</i>	color	black
<i>framed</i>	boolean	false
<i>framewidth</i>	numeric	.5bp
<i>framecolor</i>	color	black
<i>framestyle</i>	string	""
<i>shadow</i>	boolean	false
<i>shadowcolor</i>	color	black

- for each class, an option has a given default value; different classes can have different default values (but with the same types);

- once an object is defined, changing the default values has no effect; hence, default values can be changed temporarily if needed;

A container: Box

A `Box` can contain something:

- a string,
- a picture (for instance a T_EX label), or
- an object.

Containers introduce a new requirement: a *standard interface*. This is needed for the equations. For instance, one of the equations of `Box` is:

$$\begin{aligned} &.5[@\#isw, @\#ine] \\ &= .5[obj(@\#sub)ne, obj(@\#sub)sw] \end{aligned}$$

This means that the middle of the `Box` is equal to the middle of the object contained in the `Box`.

Equations can also depend on options.

And since a container does not always contain an object, the equations have to be adapted to each case.

The whole equations defined for a Box is actually:

```
ObjCode StandardEquations,
  if numeric v:
    % object
    ".5[@#isw,@#ine]=.5[obj(@#sub)ne,obj(@#sub)sw]",
  elseif (picture v) or (string v):
    ".5[@#isw,@#ine]=@#p.off", % picture offset
  fi
  if OptionValue@#("rbox_radius")>0:
    ...
  fi
  "@#ise-@#isw=(" &
    decimal (2@#a+2*OptionValue@#("dx")) & ",0)",
  "@#ine-@#ise=(0," &
    decimal (2@#b+2*OptionValue@#("dy")) & ")";
```

The value of an option can be inquired using `OptionValue`.

Standard interface and equations

The cardinal points exist in two fashions.

- First, the standard points, representing the external interface of an object. These points define how much space an object takes.

```
def StandardPoints=  
    ne,nw,sw,se,n,s,e,w,c  
enddef;
```

- Then, the standard inner points. They represent the cardinal points seen from the *inside*. These points may be used to define drawings. For instance, a square may be defined by drawing the line `ine--ise--isw--inw--cycle`.

```
def StandardInnerPoints=  
    ine,inw,isw,ise,in,is,ie,iw,ic  
enddef;
```

The importance of these points lies in the fact that they make it possible to change the bounding box (by changing the external interface) without influencing the drawing (which should be based on the internal interface).

Only initially do we have:

```
@#ine=@#ne;@#inw=@#nw;  
@#isw=@#sw;@#ise=@#se;@#in=@#n;@#is=@#s;  
@#ie=@#e;@#iw=@#w;@#ic=@#c;
```

- Returning to the Box equation:

$$\begin{aligned} &.5[@\#isw,@\#ine] \\ &= .5[obj(@\#sub)ne,obj(@\#sub)sw] \end{aligned}$$

we see that the subobject `sub` is used to its external interface (`ne`, `sw`) but that only internal interface points are used for the `Box` itself (`isw`, `ine`).

Other classes

HBox, VBox

```
newBox.a(btex Box A etex);  
newBox.b(btex Box B etex scaled \magstep3);  
newBox.c(btex Box C etex scaled \magstep2);  
newHBox.h(a,b,c);  
h.c=origin;  
drawObj(h);
```



Box A Box B Box C

Alignment can be changed. With

```
newHBox.h(a,b,c) "align(top)";
```

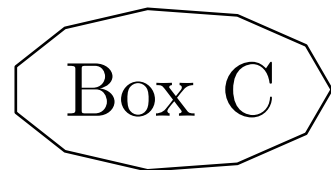
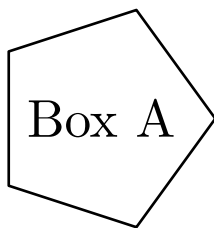
we get:



Box A Box B Box C

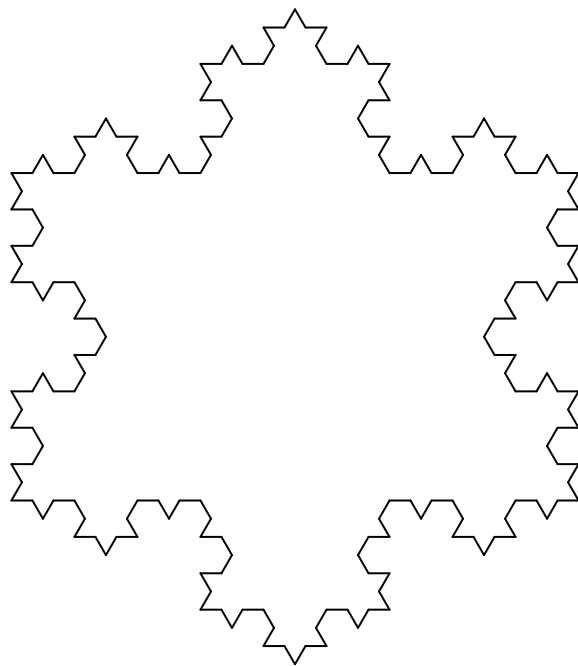
Combinations

```
newPolygon.a(btex Box A etex,5)
  "fit(false)", "polymargin(5mm)";
newBox.b(btex Box B etex scaled \magstep3);
newHRazor.ba(1cm);
newPolygon.c(btex Box C etex scaled \magstep2,11)
  "polymargin(3mm)";
newHBox.h(a,b,ba,c)
  "align(center)", "hbsep(3mm)";
h.c=origin;
drawObj(h);
```



Recursive objects

```
newVonKochFlake.a(3);  
scaleObj(a, .5);  
a.c=origin;  
drawObj(a);
```



Operations on objects

In the previous example, the Von Koch flake was scaled down.

All standard objects can be modified with the following operations:

- (translating is not relevant for floating objects);
- scaling: `scaleObj`;
- rotating: `rotateObj`;
- slanting: `slantObj`;
- reflecting: `reflectObj`;
- and any other linear transformation defined by the user.

An object can also be modified with a non-linear operation, but the user has to specify the operation.

Cloning

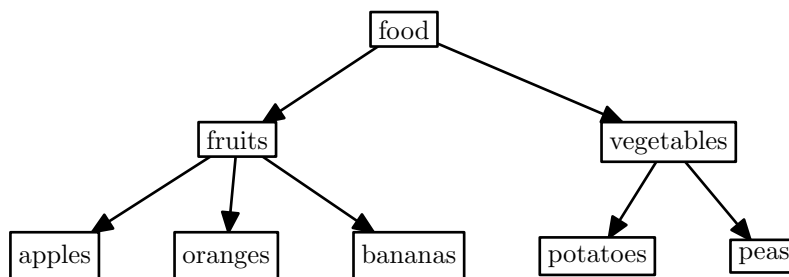
- Objects can be cloned. This is merely a matter of traversing an object and analyzing its structure and cloning its subcomponents recursively.
- A clone is a complete and independent copy of the original object.
- Partial copies would be possible, but the user has to program them.

Trees

- A tree is a complex object defined from a root and leaves.
- All these objects can belong to various classes, as long as they honor the standard interface.
- A tree is itself an object and can be used as a root or as a leaf of another tree.
- Trees can be modified by options.

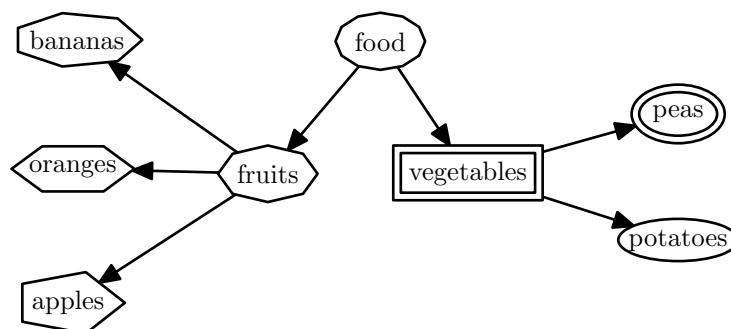
Trees: example 1

```
newBox.a(btex apples\strut etex);
newBox.b(btex oranges\strut etex);
newBox.c(btex bananas\strut etex);
newBox.f(btex fruits etex);
newTree.fruits(f)(a,b,c) "Dalign(bot)";
newBox.d(btex potatoes etex);
newBox.e(btex peas etex);
newBox.v(btex vegetables etex);
newTree.vegetables(v)(d,e)
  "Dalign(center)";
newBox.fo(btex food etex);
newTree.food(fo)(fruits,vegetables)
  "hbsep(1cm)";
scaleObj(food, .5);
food.c=origin;
drawObj(food);
```

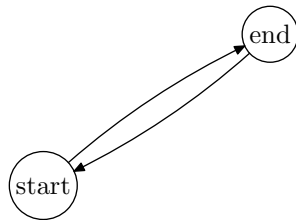


Trees: example 2

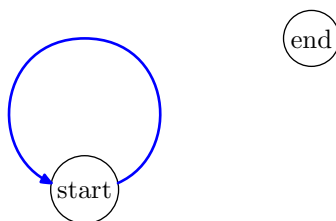
```
newPolygon.a(btex apples etex,5);
newPolygon.b(btex oranges etex,6);
newPolygon.c(btex bananas etex,7);
newPolygon.f(btex fruits etex,8);
newTree.fruits(f)(a,b,c) "Lalign(left)",
  "hideleaves(true)", "treemode(L)", "vsep(3mm)";
newEllipse.d(btex potatoes etex);
newDEllipse.e(btex peas etex);
newDBox.v(btex vegetables etex);
newTree.vegetables(v)(d,e)
  "Ralign(center)", "hideleaves(true)", "treemode(R)";
newPolygon.fo(btex food etex,12);
newTree.food(fo)(fruits,vegetables) "hsep(1cm)";
scaleObj(food,.5);
food.c=origin;
drawObj(food);
```



Connections

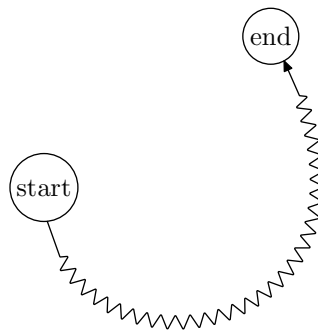


```
ncarc(a)(b);  
ncarc(b)(a);
```

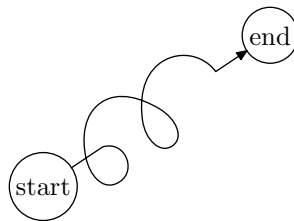


```
nccircle(a) "angleA(0)",  
"linecolor(blue)", "linewidth(1pt)";
```

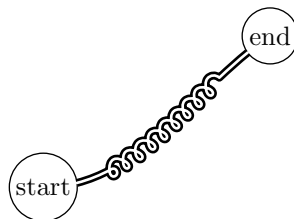
Connections (cont'ed)



```
nczigzag(a)(b) "angleA(-90)", "angleB(120)",  
"linetension(0.8)",  
"coilwidth(2mm)", "linearc(.1mm)";
```

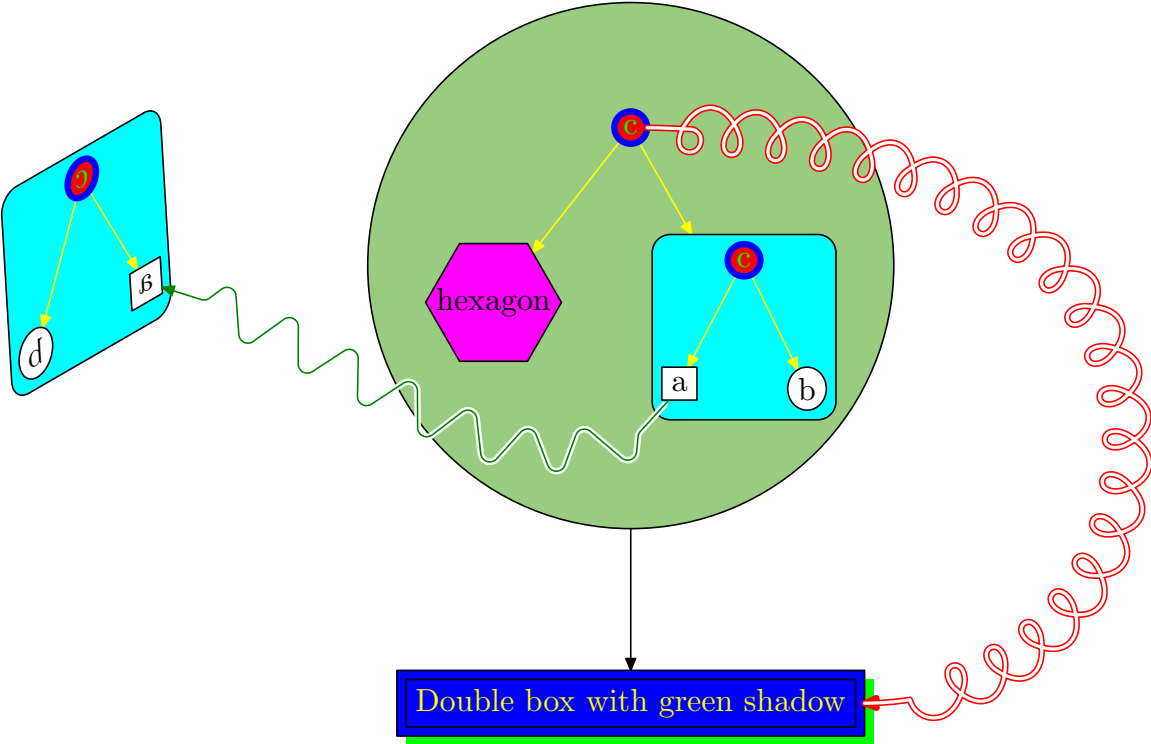


```
nccoil(a)(b);
```



```
nccoil(a)(b) "doubleline(true)", "coilwidth(2mm)",  
"angleA(0)", "arrows(-)",  
"linewidth(1pt)";
```

Complex example

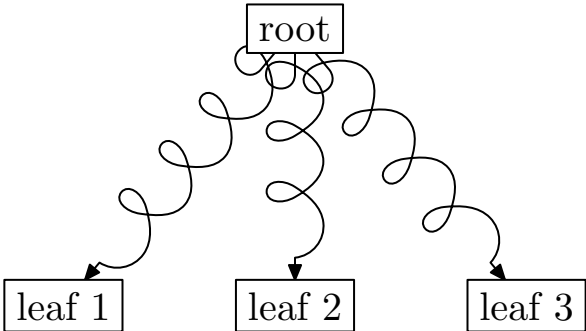
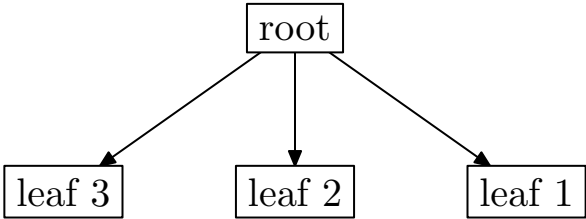
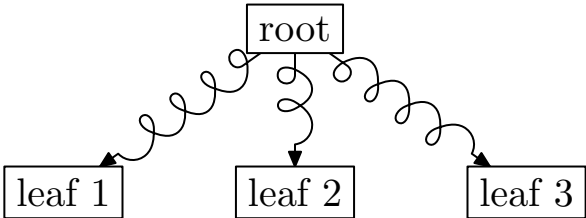


```

newBox.a("a");
newEllipse.b("b");
newEllipse.c("c")
    "filled(true)", "fillcolor(red)", "picturecolor(green)",
    "framecolor(blue)", "framewidth(2pt)";
newTree.t(c)(a,b) "linecolor((1,1,0))";
newBox.aa(t)
    "filled(true)", "fillcolor((0,1,1))", "rbox_radius(2mm)";
aa.c=origin;
newHexagon.xa("hexagon")
    "fit(false)", "filled(true)", "fillcolor((1,0,1))";
newEllipse.xc("c")
    "filled(true)", "fillcolor(red)", "picturecolor(green)",
    "framecolor(blue)", "framewidth(2pt)";
newTree.xt(xc)(xa,aa) "linecolor((1,1,0))";
newCircle.xaa(xt) "filled(true)", "fillcolor((.6,.8,.5))";
newDBox.db(btex Double box with green shadow etex)
    "shadow(true)", "shadowcolor(green)",
    "filled(true)", "fillcolor(blue)",
    "picturecolor((1,1,0))";
newTree.nt(xaa)(db);
drawObj(nt);
nccoil(xc)(db) "angleA(0)", "angleB(180)",
    "coilwidth(5mm)", "linetension(0.8)", "linecolor(red)",
    "doubleline(true)", "posB(e)";
duplicateObj(dt,aa);
reflectObj(dt,origin,up);
slantObj(dt,.5);
rotateObj(dt,30);
dt.c=nt.c-(6cm,-1cm);
drawObj(dt);
nczigzag(a)(treepos(obj(dt.sub))(1))
    "angleA(-120)", "coilwidth(7mm)",
    "linecolor(.5green)", "linearc(1mm)",
    "border(2pt)";

```

Object extraction



Comparison with other packages

- `(r)boxes.mp` is subsumed by `MetaObj`;
- `PSTricks`: many of its features, especially connections, and options, are taken over in `MetaObj`;
- `fancybox` is subsumed by `MetaObj` (for the frame features).

Extensions

- new classes can be added easily, possibly building some of them out of existing classes;
- layers are not implemented but could be implemented;
- a $\text{T}_{\text{E}}\text{X}$ interface would solve the problem of syntax constraints.

Possible T_EX interface

MetaObj code:

```
setObjectDefaultOption("Tree")("treemode")("D");
setCurveDefaultOption("arrows")("drawarrow");
t:=T_(new_Polygon_(btex root etex)(4)("name(top)"))
      (new_Box_(btex x etex)("framed(false)","name(lx)"),
       new_Box_(btex y etex)("framed(false)","name(ly)"),
       new_Box_(btex z etex)("framed(false)","name(lz)"))
      ("edge(none)","vsep(1.5cm)");
ncbar.Obj(t)("top")("lx") "angleA(180)","armA(1cm)";
ncbar.Obj(t)("top")("ly");
ncbar.Obj(t)("top")("lz") "angleA(0)","armA(1cm)";
Obj(t).c=origin;
draw_Obj(t);
```


MetaObj hidden by a (not yet existing) T_EX interface:

```
\begin{metaobj}
\mosetO{Tree}{treemode=D}
\mosetC{arrows=drawarrow}
\setObj{t}{\Tree{\Polygon{root}{4}[name=top]}
            {\Box{x}[framed=false,name=lx],
             \Box{y}[framed=false,name=ly],
             \Box{z}[framed=false,name=lz]}
            [edge=none,vsep=1.5cm]}
\ncbar[t]{top}{lx}[angleA=180,armA=1cm]
\ncline[t]{top}{ly}
\ncbar[t]{top}{lz}[angleA=0,armA=1cm]
\pos{t.c}{origin}
\draw{t}
\end{metaobj}
```

Conclusions

- MetaObj was an experimental project aimed at exploring the feasibility of 'high-level structures' in MetaPost;
- it appears that a lot is possible, and actually much more, since interesting concepts, such as layers, have not been implemented; MetaPost is actually vastly underused;
- we have a system with a functional approach and keeping the interesting declarative features of MetaPost;
- MetaPost also appears quite robust, and the MetaObj code sometimes gets very tricky (it also taught me a lot!);

- but due to the MetaPost syntax constraints, using MetaObj is still complex; the syntax idiosyncrasies could be hidden within $\text{T}_{\text{E}}\text{X}$;
- the structure of an object could also be modelled outside of MetaPost, but then using $\text{T}_{\text{E}}\text{X}$ may be difficult;

More information:

CTAN:

`graphics/metapost/contrib/macros/metaobj`