Nodes

Alan Braslau

The graphical representation of textual diagrams is presented.

Introduction

An earlier version of this article was published in `TUGboat`. (Braslau, Hamid, & Hagen, 2018)

The graphical representation of textual diagrams is a very useful tool in the communication of ideas. In category and topos theory, for example, many key concepts, formulas, and theorems are expressed by means of *commutative diagrams*; these involve objects and arrows between them. Certain concepts discovered by category theory, such as *natural transformations*, are becoming useful in areas outside of mathematics and natural science, e.g., in philosophy. To make category and topos methods usable by both specialists and non-specialists, commutative diagrams are an indispensable tool.[1]The use of nodal diagrams is not limited to category theory, for they may represent a flow diagram (of a process, for example), a chemical reaction sequence or pathways, or that of phases and phase transitions, a hierarchical structure (of anything), a timeline or sequence of events or dependencies, a tree of descendance or ascendance, etc.

The basic units of a node-based diagram include *node objects*, each attached to some point (= the *node*) in some spatial relationship. Note that to a single node might be associated a set of objects. Given a node, it also stands in a spatial relation to some other node. The spatial relationship between the set of nodes of a diagram need not be in a regular network, although they quite often are. Note that the spatial relationship between nodes is graphical and may represent, e.g., a temporal or logical relationship, or a transformation of one object into another or into others (one interesting example might be that representing cell division or, mitosis).

Given a spatial relation between any two nodes, a node-based diagram often includes some *path segment* or segments (such as arrows or other curves) between two given nodes that *relates* them. Each path segment may be augmented by some textual or graphical label.

A very simple example of a node diagram is shown in Figure 1.

$$A \longrightarrow B$$

Figure 1.

More precisely, a *node* is a point of intersection or branching of paths, often a point on a regular lattice. (The nodes of the above diagram are the two endpoints of a

[1] For many examples of formal and informal commutative diagrams, see Lawvere and Schanuel (2009).

straight line segment.) Sometimes, however, a node might be a single point as in an *identity map* of category theory, referring to itself:[2]

Figure 2.

In this article we discuss a new MetaPost module designed for handling node-based graphics as well as a derivative simple ConTEXt interface. To illustrate, the code producing A⟶B could be, in MetaPost and the ConTEXt interface respectively:

MetaPost

```
\startMPcode
  draw node(0,"A") ;
  draw node(1,"B") ;
  drawarrow fromto(0,1) ;
\stopMPcode
```

ConTEXt

```
\startnodes [dx=1.5cm]
  \placenode    [0,0] {A}
  \placenode    [1,0] {B}
  \connectnodes [0,1] [alternative=arrow]
\stopnodes
```

drawing an arrow from A to B (or from node 0 to node 1): A⟶B

1. The MetaPost code shown above has been slightly simplified, as will be seen later.
2. The ConTEXt interface as used here is limited and will be explained a little later.

For beginners or casual users of ConTEXt —including those who might be intimidated by MetaPost syntax— the ability to construct simple diagrams by means of standard ConTEXt syntax is very helpful. For those who have tried the ConTEXt interface and/or want to draw more advanced diagrams, the MetaPost module is much more powerful and flexible.

[2] The standard arrowhead in MetaPost is a simple triangle, whose length and angle can be adjusted. MetaFun provides further options, allowing this arrowhead to be barbed or dimpled. In the present article, we use the setting: `ahvariant := 1 ;` The loop-back arrow paths used here deviate from a circular segment, becoming ellipsoidal, through the value `node_loopback_yscale := .7 ;` These are all set, of course, between a `\startMPinitializations` ... `\stopMPinitializations` pair.

MetaPost

MetaPost is a vector-graphics language which calls upon TEX to typeset text (such as labels); in ConTEXt, furthermore, MetaPost is integrated natively through the library MPlib as well as the macro package MetaFun. The tight integration of ConTEXt and MetaPost provides advantages over the use of other, external graphics engines. These advantages include ease of maintaining coherence of style, as well as extensive flexibility without bloat. MetaPost has further advantages over most other graphics engines, including a very high degree of precision as well as the possibility to solve certain types of algebraic equations. This last feature is rarely used but should not be overlooked.

It is quite natural in MetaPost to locate our node objects along a path or on differing paths. This is a much more powerful concept than merely locating a node at some pair of coordinates, e.g., on a square or a rectangular lattice, for example (as in a table). Furthermore, these paths may be in three dimensions (or more); of course the printed page will only involve some projection onto two dimensions. Nor are the nodes restricted to location on the points defining a path: they may have, as index, any *time* along a given path `p` ranging from the first defining point ($t = 0$) up to the last point of that path ($t \leq \texttt{length(p)}$), the number of defining points of a path.[3]

Given a path `p`, nodes are defined (implicitly) as `picture` elements: `picture p.pic[] ;` This is a pseudo-array where the square brackets indicates a set of numerical tokens, as in `p.pic[0]` or `p.pic[i]` (for `i=0`), but also `p.pic0`. This number need not be an integer, and `p.pic[.5]` or `p.pic.5` (not to be confused with `p.pic5`) are also valid. These picture elements are taken to be located relative to the path `p`, with the index `t` corresponding to a time along the path, as in `draw p.pic[t] shifted point t of p;` (although it is not necessary to draw them in this way). This convention allows the nodes to be oriented and offset with respect to the path in an arbitrary manner.

Note that a path can be defined, then nodes placed relative to this path. Or the path may be declared but remain undefined, to be determined only after the nodes are declared. In yet another possibility, the path may be adjusted as needed, as a function of whatever nodes are to be occupied. This will be illustrated through examples further down.

A few simple examples

Let's begin with illustration of a typical commutative diagram from category theory. Although it may appear trivial, this example helps to introduce MetaPost syntax. At the same time, a large part of the idea behind this module is to facilitate use of this system without having to learn much MetaPost.

A path is drawn as well as the points defining the path.

[3] Note that the time of a cyclic path is taken modulo the length of the path, that is t outside of the range $[0, \texttt{length(p)}]$ will return the first or the last point of an open path, but will "wrap" for a closed path.

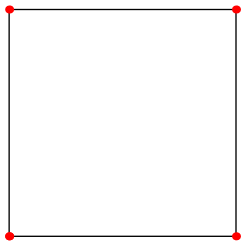

Figure 3.

```
\startMPcode
  path p ; p := fullsquare scaled 3cm ;
  draw p ;
  for i=0 upto length p:
    draw point i of p
      withpen pencircle scaled dotlabeldiam
      withcolor red ;
  endfor ;
\stopMPcode
```

Given the named path `nodepath`, we can now define and draw nodes as well as connections between them (see Figure 4):

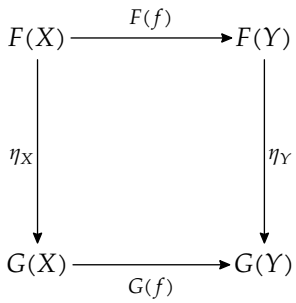

Figure 4. Drawn using MetaPost interface

```
\startMPcode
  clearnodepath ;
  nodepath = p ;
  draw node(0,"\node{$G(X)$}") ;
  draw node(1,"\node{$G(Y)$}") ;
  draw node(2,"\node{$F(Y)$}") ;
  draw node(3,"\node{$F(X)$}") ;
  drawarrow fromto.bot(0,0,1,"\smallnode{$G(f)$}") ;
  drawarrow fromto.top(0,3,2,"\smallnode{$F(f)$}") ;
  drawarrow fromto.rt (0,2,1,"\smallnode{$_Y$}") ;
  drawarrow fromto.lft(0,3,0,"\smallnode{$_X$}") ;
\stopMPcode
```

In working with MetaPost, it is good practice to reset or clear a variable using the directive `save` for the *suffix* (or variable name) `nodepath` contained in the directive `clearnodepath` (defined as "`save nodepath ; path nodepath`"). The macros used here rely on the creation of certain internal variables and may not function correctly if the variable structure is not cleared. Indeed, any node may contain a combination of picture elements, added successively, so it is very important to `save` the variable, making its use local, rather than global. This point is particularly true with ConTEXt, where a single MPlib instance is used and maintained over multiple runs.

The ConTEXt directives `\startMPcode` ... `\stopMPcode` include grouping (MetaPost `begingroup ;`... `endgroup ;`) and the use of `save` (in `clearnodepath`) will make the suffix `nodepath` local to this code block. In the code for Figures 3 and 4, the path `p` itself is not declared local (through the use of a `save`); it therefore remains available for other MetaPost code blocks. We cannot do this with the default suffix name `nodepath` without undesired consequences.

The directive `clearnodepath` used in the example above, much like the MetaPost command `clearxy` clearing the `(x,y)` pair also known as `z`, uses `save` to clear the default suffix `nodepath`.

Note that one should not confuse the MetaPost function `node()` with the ConTEXt command `\node{}`, defined as follows:

```
\defineframed
  [node]
  [frame=off,offset=1pt]

\defineframed
  [smallnode]
  [node]
  [foregroundstyle=small]
```

`\node{}` places the text within a ConTEXt frame (with the frame border turned-off), whereas the MetaPost function `node(i,"…")` sets and returns a picture element associated with a point on path `nodepath` indexed by its first argument. The second argument here is a string that gets typeset by TEX. (The use of `\node{}` adds an `offset`).

By default, the MetaPost function `fromto()` returns a path segment going between two points of the path `nodepath`. The first argument (`0` in the example above) can be used as a displacement to skew the path away from a straight line (by an amount in units of the straight path length). The last argument is a string to be typeset and placed midpoint of the segment. The suffix appended to the function name gives an offset around this halfway point. This follows standard MetaPost conventions.

It is important to draw or declare the nodes *before* drawing the connections, using `fromto()`, in order to be able to avoid overlapping symbols, as one notices that the

arrows drawn in the example above begin and end on the border of the frame (or bounding box) surrounding the node text. This would of course not be possible if the arrow were to be drawn before this text was known.

As will be seen further on, one can actually specify the use of any defined path, without restriction to the built-in name `nodepath` that is used by default. Furthermore, a function `fromtopaths()` can be used to draw segments connecting any two paths which may be distinct. This too will be illustrated further on.

The ConTEXt syntax for the current example looks like this:

```
\startnodes [dx=3cm,dy=3cm,alternative=arrow]
  \placenode [0,0] {\node{$G(X)$}}
  \placenode [1,0] {\node{$G(Y)$}}
  \placenode [1,1] {\node{$F(Y)$}}
  \placenode [0,1] {\node{$F(X)$}}
  \connectnodes [0,1] [label={\smallnode{$G(f)$}},
       position=bottom]
  \connectnodes [3,2] [label={\smallnode{$F(f)$}},
       position=top]
  \connectnodes [2,1] [label={\smallnode{$_Y$}},
       position=right]
  \connectnodes [3,0] [label={\smallnode{$_X$}},
       position=left]
\stopnodes
```

$$\begin{tikzcd} F(X) \arrow[r, "F(f)"] \arrow[d, "\eta_X"'] & F(Y) \arrow[d, "\eta_Y"] \\ G(X) \arrow[r, "G(f)"'] & G(Y) \end{tikzcd}$$

Figure 5. Drawn using ConTEXt interface

This follows the more classic (and limited) approach of placing nodes on the coordinates of a regular lattice, here defined as a 3 cm square network.[4] The arguments are then (x, y) coordinates of this lattice and the nodes are indexed 0, 1, 2, ... in the order that they are drawn. These are used as reference indices in the commands `\connectnodes` (rather than requiring two *pairs* of coordinates); see Figure 6. This might seem a bit confusing at first view, but it simplifies things in the end, really!

[4] The lattice can be square (dx = dy), rectangular (dx ≠ dy), or oblique (through `rotation` ≠ 90).

[5] For variety, a rectangular oblique lattice is drawn in Figure 6.

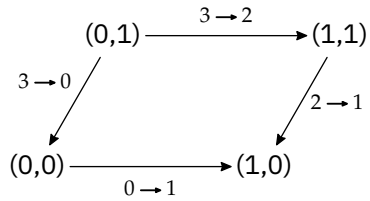

Figure 6. Coordinates and indices.[5]

To avoid such confusion, in particular when drawing complicated diagrams containing many nodes, the ConTEXt interface allows the use of a `reference` or tag, assigning a symbolic name to a numbered node. The example of Figure 5 can be redrawn a little more verbosely as:

```
\startnodes [dx=3cm,dy=3cm,alternative=arrow]
  \placenode [0,0] [reference=GX] {\node{$G(X)$}}
  \placenode [1,0] [reference=GY] {\node{$G(Y)$}}
  \placenode [1,1] [reference=FY] {\node{$F(Y)$}}
  \placenode [0,1] [reference=FX] {\node{$F(X)$}}
  \connectnodes [GX,GY] [label={\smallnode{$G(f)$}},
       position=bottom]
  \connectnodes [FX,FY] [label={\smallnode{$F(f)$}},
```

```
       position=top]
  \connectnodes [FY,GY] [label={\smallnode{$_Y$}},
       position=right]
  \connectnodes [FX,GX] [label={\smallnode{$_X$}},
       position=left]
\stopnodes
```

Notice that, like for all ConTEXt macros, one never mixes a comma-separated list of arguments (`[0,0]`) with a key=value list (i.e. `[reference=GX]`). Symbolic references are very useful for longer, more complicated diagrams; additionally, this easily allows modification such as the addition of nodes without having to keep track of counting the order in which they were drawn.

An identity map, as shown in Figure 2, earlier, and, below, in Figure 7 is achieved by connecting a node to itself:

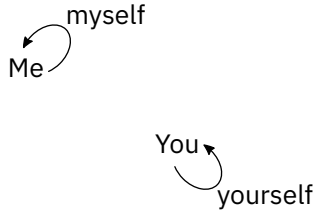

Figure 7. Identity maps

```
\startnodes [dx=2cm,dy=1cm,alternative=arrow]
  \placenode [0,0]  {\node{Me}}
  \placenode [1,-1] {\node{You}}
  \connectnodes [0,0] [offset=.75cm,position=topright,
                       label=myself]
  \connectnodes [1,1] [offset=.75cm,position=bottomright,
                       label=yourself]
\stopnodes
```

The scale (diameter) of the circular loop-back is set by the keyword `offset=` (normally used to curve or bow-away a path connecting nodes from the straight-line segment between them), and the `position=` keyword sets its orientation.

Let us now consider the following code which illustrates the MetaFun operator `crossingunder`[6] (see Figure 8). The `nodepath` indices are put into variables `A`, `B`, `C`, and `D`, thus simplifying the code.

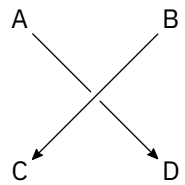

Figure 8. A→D under B→C

```
\startMPcode
clearnodepath ;
nodepath = fullsquare scaled 2cm ;
save A ; A = 3 ; draw node(A,"\node{A}") ;
save B ; B = 2 ; draw node(B,"\node{B}") ;
save C ; C = 0 ; draw node(C,"\node{C}") ;
save D ; D = 1 ; draw node(D,"\node{D}") ;

drawarrow fromto(0,B,C) ;
drawarrow fromto(0,A,D) crossingunder fromto(0,B,C) ;
\stopMPcode
```

Another illustration of the `crossingunder` operator in use is shown in figure 9. Because the diagrams are all defined and drawn in MetaPost, one can easily use the power of MetaPost to extend a simple node drawing with any kind of graphical decoration.

[6] The operator `crossingunder` is of such general use that it has been added to the MetaFun base.

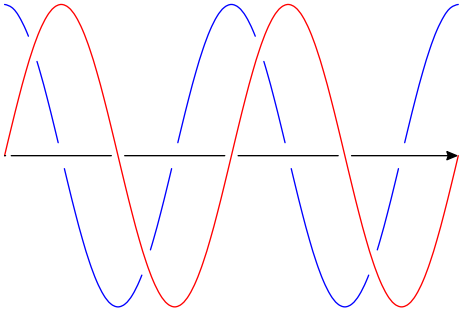

Figure 9. `crossingunder`

This brings up an important point that has limited the development of a full-featured ConTEXt module up to now. A pure MetaPost interface affords much more flexibility than can be conveniently reduced to a set of TEX macros; the ConTEXt interface has been written to maintain only basic functionality.[7]

Cyclic diagrams

For a somewhat more complicated example, let us consider the representation of a catalytic process such as that given by Krebs. (Krebs, 1946) The input is shown coming into the cycle from the center of a circle; the products of the cycle are spun-off from the outside of the circle. We start by defining a circular path where each point corresponds to a step in the cyclic process. Our example will use six steps (see Figure 10).

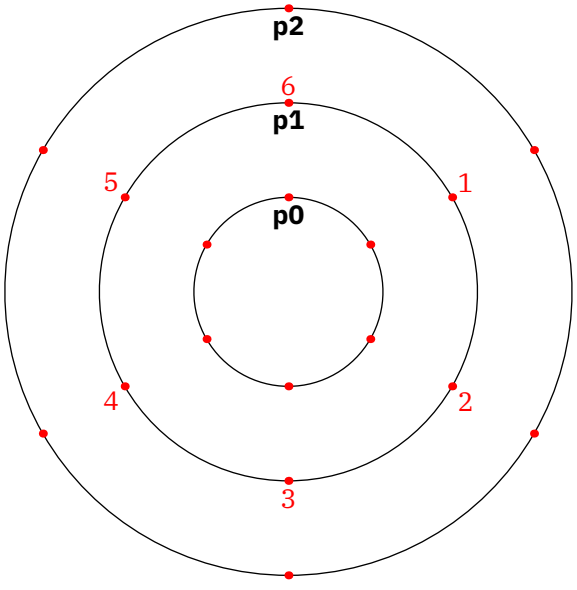

Figure 10. The paths that we will use for the anchoring of nodes.

[7] One can use `\nodeMPcode{}` to inject arbitrary MetaPost code within a `\startnode` ... `\stopnode` pair, although in this example one is probably better off using the straight MetaPost interface.

We also want to define a second circular path with the same number of points at the interior of this first circle for the input, and a third circular path at the exterior for the output.

The code is as follows:

```
\startMPcode
  save p ; path p[] ;
  % define a fullcircle path with nodes at 60° (rather than 45°)
  p1 := (for i=0 step 60 until 300:
           dir(90-i) ..
         endfor cycle) scaled 2.5cm ;
  p0 := p1 scaled  .5 ;
  p2 := p1 scaled 1.5 ;

  for i=0 upto 2:
    draw p[i] ;
    label.bot("\bf p" & decimal i, point 0 of p[i]) ;
    for j=1 upto length p[i]:
      if i=1:
        dotlabel.autoalign(angle point j of p[i])
                     (decimal j, point j of p[i])
          withcolor red ;
      else:
        draw point j of p[i]
          withpen pencircle scaled dotlabeldiam
          withcolor red ;
      fi
    endfor
  endfor
\stopMPcode
```

[`autoalign()` is a feature defined within MetaFun.]

Nodes will then be drawn on each of these three circles and arrows will be used to connect these various nodes, either on the same path or else between paths.

The MetaPost function `fromto()` is used to give a path segment that points from one node to another. It *assumes* the path `nodepath`, and in fact calls the function `fromtopaths` that explicitly takes path names as arguments. That is, `fromto (d, i, j, …)` is equivalent to `fromtopaths (d, nodepath, i, nodepath, j, …)`.

As stated above, this segment can be a straight line or else a path that can be bowed-away from this straight line by a transverse displacement given by the function's first argument (given in units of the straight segment length). When both nodes are located on a single, defined path, this segment can be made to lie on or follow this path, such as one of the circular paths defined above. This behavior is obtained by

using any non-numeric value (such as `true`) in place of the first argument. Of course, this cannot work if the two nodes are not located on the same path.

The circular arc segments labeled *a–f* are drawn on figure 11 using the following:

```
drawarrow fromtopaths.urt(true,p1,0,p1,1,"\greennode{a}") ;
```

for example, where `\greennode` is a frame that inherits from `\node`, changing style and color:

```
\defineframed
  [greennode]
  [node]
  [foregroundcolor=darkgreen,
   foregroundstyle=italic]
```

The bowed-arrows feeding into the cyclic process and leading out to the products, thus between different paths, from the path `p0` to the path `p1` and from the path `p1` to the path `p2`, respectively, are drawn using the deviations `+3/10` and `-1/10` (to and from half-integer indices, thus mid-step, on path `p1`):

```
drawarrow fromtopaths( 3/10,p0,0,p1,0.5) withcolor .6white ;
drawarrow fromtopaths(-1/10,p1,0.5,p2,1) withcolor .6white ;
```

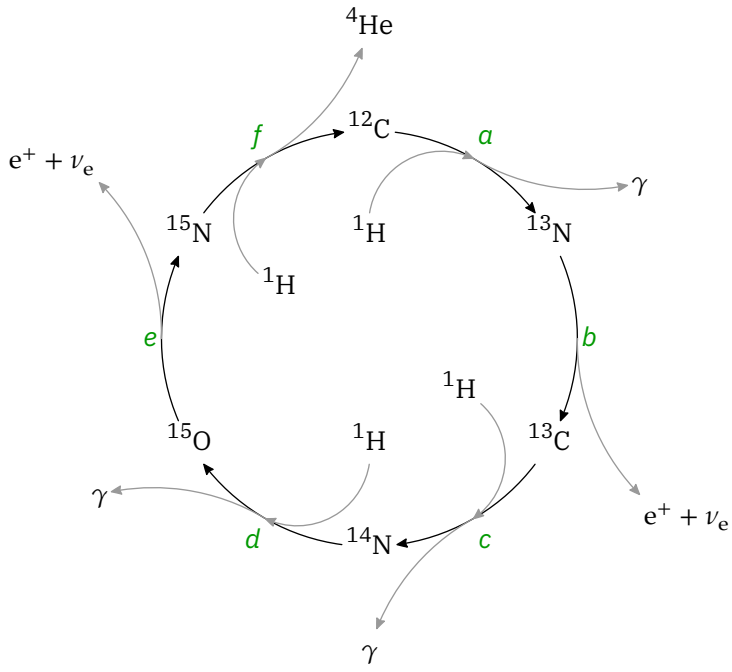

Figure 11. The Bethe cycle for the energy production in stars (1939) in a Krebs representation of a catalytic process (1946).

A lesson in MetaPost

An ‘array’ of paths is declared through `path p[] ;` it is not a formal array, but rather a syntactic definition of a collection of path variables `p0`, `p1`, ... each of whose name is prefixed with the tag “p” followed by any number, not necessarily an integer (e.g., `p3.14` is a valid path name). The syntax allows enclosing this “index” within square brackets, as in `p[0]` or, more typically, `p[i]`, where `i` would be a numeric variable or the index of a loop. Note that the use of brackets is required when using a negative index, as in `p[-1]` (for `p-1` is interpreted as three tokens, representing a subtraction). Furthermore, the variable `p` itself, would here be a numeric (by default), so `p[p]` would be a valid syntactic construction! One could, additionally, declare a set of variables `path p[][] ;` and so forth, defining also `p[0][0]` (equivalently, `p0 0`) for example as a valid path, coexisting with yet different from the path `p0`.

MetaPost also admits variable names reminiscent of a structure: `picture p.pic[] ;` for example is used internally in the `node` macros, but this becomes `picture p[]pic[] ;` when using a path ‘array’ syntax. These variable names are associated with the suffix `p` and become all undefined by `save p ;`.

Putting it together

What follows is simple example of a natural transformation, discovered and articulated in the course of a philosophical research project (by Idris Samawi Hamid). Figure 12 represents what is called the Croce Topos [named after the Italian philosopher Benedetto Croce (1866–1952)]:

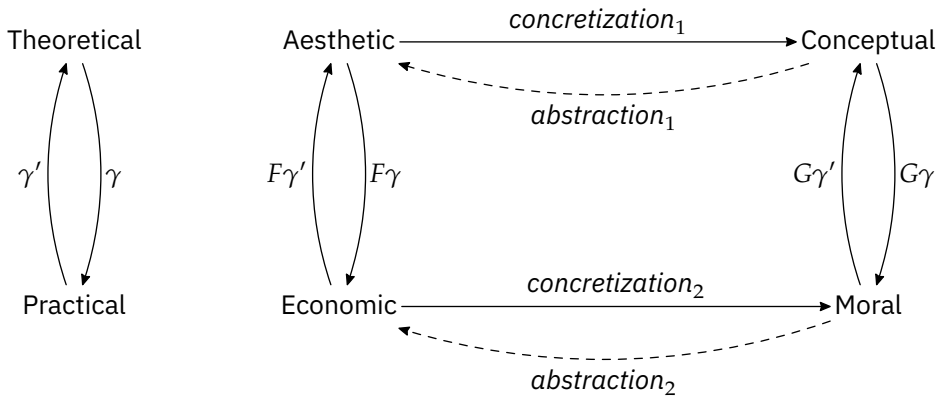

Figure 12. A representation of the Croce Topos

Here we use the ConTEXt interface to the node package:

```
\startnodes [dx=3.5cm,dy=3.5cm,alternative=arrow]
  \placenode [0,0] {\node{Practical}}
  \placenode [1,0] {\node{Economic}}
  \placenode [3,0] {\node{Moral}}
  \placenode [3,1] {\node{Conceptual}}
  \placenode [1,1] {\node{Aesthetic}}
```

```
  \placenode [0,1] {\node{Theoretical}}

  \connectnodes [5,0]
      [offset=.1,position=right,label={\node{$$}}]
  \connectnodes [0,5]
      [offset=.1,position=left, label={\node{$'$}}]
  \connectnodes [4,1]
      [offset=.1,position=right,label={\node{$F$}}]
  \connectnodes [1,4]
      [offset=.1,position=left, label={\node{$F'$}}]
  \connectnodes [3,2]
      [offset=.1,position=right,label={\node{$G$}}]
  \connectnodes [2,3]
      [offset=.1,position=left, label={\node{$G'$}}]

  \connectnodes [4,3]
      [position=top,   label={\node{\it concretization$_1$}}]
  \connectnodes [3,4]
      [position=bottom,label={\node{\it abstraction$_1$}},
       offset=.1,option=dashed]
  \connectnodes [1,2]
      [position=top,   label={\node{\it concretization$_2$}}]
  \connectnodes [2,1]
      [position=bottom,label={\node{\it abstraction$_2$}},
       offset=.1,option=dashed]
\stopnodes
```

Tree diagrams

The tree diagram shown in Figure 13 is drawn using four paths, each one defining a row or generation in the branching. The definition of the spacing of nodes was crafted by hand and is somewhat arbitrary: 3.8, 1.7, and 1 for the first, second and third generations. This might not be the best approach, but this is how I was thinking when I first created this figure.

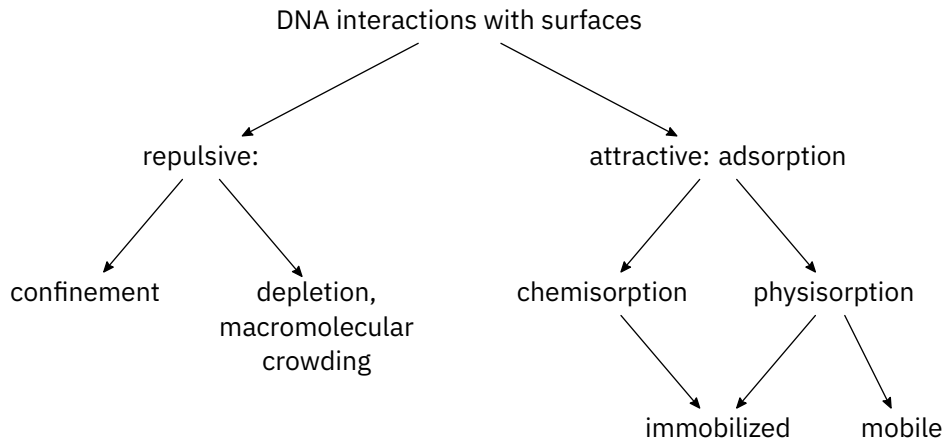

Figure 13.

Ultimately, one can do better by allowing MetaPost to solve the relevant equations and to determine this spacing automatically. Because it is a somewhat advanced procedure, this approach will be first illustrated through a very simple example of a diagram where the nodes will be placed on a declared but undefined path:

```
save p ; % path p ;
```

The `save p ;` assures that the path is undefined. This path will later be defined based on the contents of the nodes and a desired relative placement. In fact, it is not even necessary to declare that the suffix will be a path, as the path will be declared and automatically built once the positions of all the nodes are determined. To emphasize this point, the `path` declaration above is commented-out.

Warning: Solving equations in MetaPost can be non-trivial for those who are less mathematically inclined. One needs to establish a coupled set of equations that is solvable: that is, fully but not over-determined.

A few helper functions have been defined: `makenode()` returns a suffix (variable name) corresponding to the node's position. The first such node can be placed at any finite point, for example the drawing's origin. The following nodes can be placed in relation to this first node:

```
save nodepath ;
save first, second, third, fourth ;
pair first, second, third, fourth ;
first.i  = 0 ; first  = makenode(first.i, "\node{first}") ;
second.i = 1 ; second = makenode(second.i,"\node{second}") ;
third.i  = 2 ; third  = makenode(third.i, "\node{third}") ;
fourth.i = 3 ; fourth = makenode(fourth.i,"\node{fourth}") ;

first  = origin ;
```

```
second = first
       + betweennodes.urt(nodepath,first.i,nodepath,second.i,
                          whatever) ;
third  = second
       + betweennodes.lft(nodepath,second.i,nodepath,third.i,
                          whatever) ;
fourth = third
       + betweennodes.bot(nodepath,fourth.i,nodepath,first.i,
                          3ahlength) ;
```

The helper function `betweennodes()` returns a vector pointing in a certain direction, here following the standard MetaPost suffixes: `urt`, `lft`, and `bot`, that takes into account the bounding boxes of the contents of each node, plus an (optional) additional distance (here given in units of the arrow-head length, `ahlength`). Using the keyword `whatever` tells MetaPost to adjust this distance as necessary. The above set of equations is incomplete as written, so a fifth and final relation needs to be added; the fourth node is also to be located directly to the left of the very first node:[8]

```
fourth = first
       + betweennodes.lft(nodepath,fourth.i,nodepath,first.i,
                          3ahlength) ;
```

Note that the helper function `makenode()` can be used as many times as needed; if given no content, only returning the node's position. Additional nodes can be added to this diagram along with appropriate relational equations, keeping in mind that the equations must, of course, be solvable. This last issue is the one challenge that most users might face. The function `node()` that was used previously and returning a picture element to be drawn itself calls the function `makenode()`, used here. The nodes have not yet been drawn:

```
for i = first.i, second.i, third.i, fourth.i :
  draw node(i) ;
  drawarrow fromto(0,i,i+1) ;
endfor
```

This results in Figure 14. The path is now defined as one running through the position of all of the defined nodes, and is cyclic.

[8] Equivalently, we could declare that the first node located to the right of the fourth node: `first = fourth + betweennodes.rt (nodepath, first.i, nodepath, fourth.i, 3ahlength) ;`

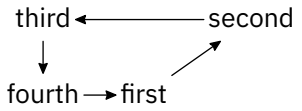

Figure 14.

Using this approach, that of defining but not drawing the nodes until a complete set of equations defining their relative positions has been constructed, imposes several limitations. First, the nodes are expected to be numbered from 0 up to n, continuously and without any gaps for each defined path. This is just an implicit, heuristic convention of the path construction. Second, when finally defining all the nodes and their positions, the path needs to be constructed. A function, `makenodepath(p)` ; accomplishes this; it gets implicitly called (once) upon the drawing of any `node()` or connecting `fromto`. Of course, `makenodepath()` can always be called explicitly once the set of equations determining the node positions is completely defined.

We once again stress that the writing of a solvable yet not over-determined set of equations can be a common source of error for many MetaPost users.[9]

Another such example is the construction of a simple tree of descendance or family tree. There are many ways to draw such a tree; in figure 15 we will show only three generations.

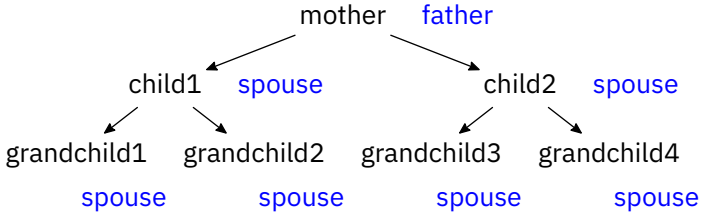

Figure 15. A tree of descendance

We leave it as an exercise to the reader to come-up with the equations used to determine this tree (one can look at source of this document, if necessary).

The requisite set of equations could be hidden from the user wishing to construct simple, pre-defined types of diagrams. However, such cases would involve a loss of generality and flexibility. Nevertheless, the ConTEXt-Nodes module *could* be extended in the future to provide a few simple models. One might be a branching tree structure, although even the above example (as drawn) does not easily fit into a simple, general model.

A user on the ConTEXt mailing list asked if it is possible to make structure trees for English sentences with categorical grammar, an example of which is shown in Figure 16.

[9] The generous use of descriptive variables as we try to illustrate in the examples here can help tremendously in keeping track of multiple equations.

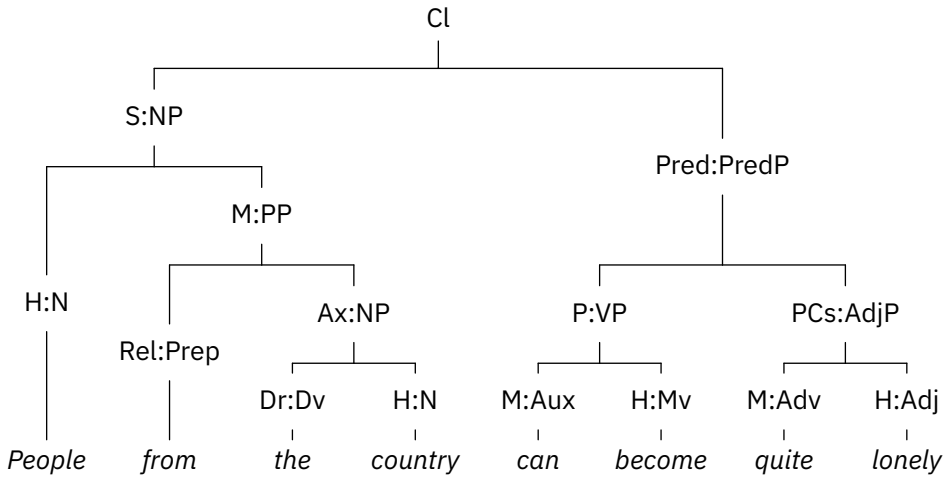

Figure 16. A categorical grammer structure tree

Here, I chose to define a series of parallel paths, one per word, with one path terminating whenever it joins another path (or paths) at a common parent. Naturally, labeling each branch of the tree structure requires a knowledge of the tree structure. The code is not short, but hopefully it is mostly clear.

```
\startMPcode
  save p ; path p[] ;
  save n ; n = 0 ;
  % rather than parsing a string, we can use "suffixes":
  forsuffixes $=People,from,the,country,can,become,quite,lonely :
    p[n] = makenode(p[n],0,"\node{\it" & (str $) & "}")
         = (n,0) ; % we work first with unit paths.
    n := n + 1 ;
  endfor
  save u ; u := MakeupWidth/n ;

  % build upward tree

  vardef makeparentnode(text t) =
    save i, xsum, xaverage, ymax ;
    i = xsum = 0 ;
    forsuffixes $ = t :
      clearxy ; z = point infinity of $ ;
      xsum := xsum + x ;
      if unknown ymax :
        ymax = y ;
      elseif y > ymax :
        ymax := y ;
      fi
```

```
      i := i + 1 ;
    endfor
    xaverage = xsum / i ;
    ymax := ymax + 1 ;
    forsuffixes $ = t :
      clearxy ; z = point infinity of $ ;
      $ := $ & z -- (x,ymax)
        if i>1 : -- (xaverage,ymax) fi ;
    endfor
  enddef ;

  makeparentnode(p2,p3) ;
  makeparentnode(p4,p5) ;
  makeparentnode(p6,p7) ;
  makeparentnode(p1,p2) ;
  makeparentnode(p0,p1) ;
  makeparentnode(p4,p6) ;
  makeparentnode(p0,p4) ;
  makeparentnode(p0) ;

  % the paths are all defined
  % but need to be scaled.

  for i=0 upto n-1 :
    p[i] := p[i] xyscaled (u,.8u) ;
    draw node(p[i],0) ;
  endfor

  save followpath ; boolean followpath ; followpath = true ;

  draw fromtopaths(followpath,p0,0,p0,1,"\node{H:N}") ;
  draw fromtopaths(followpath,p1,0,p1,1,"\node{Rel:Prep}") ;
  draw fromtopaths(followpath,p2,0,p2,1,"\node{Dr:Dv}") ;
  draw fromtopaths(followpath,p3,0,p3,1,"\node{H:N}") ;
  draw fromtopaths(followpath,p4,0,p4,1,"\node{M:Aux}") ;
  draw fromtopaths(followpath,p5,0,p5,1,"\node{H:Mv}") ;
  draw fromtopaths(followpath,p6,0,p6,1,"\node{M:Adv}") ;
  draw fromtopaths(followpath,p7,0,p7,1,"\node{H:Adj}") ;

  draw fromtopaths(followpath,p1,1,p1,2) ;
  draw fromtopaths(followpath,p2,3,p2,4) ;
  draw fromtopaths(followpath,p1,2,p1,3,"\node{M:PP}") ;
  draw fromtopaths(followpath,p2,1,p2,2) ;
  draw fromtopaths(followpath,p3,1,p3,2) ;
  draw fromtopaths(followpath,p2,2,p2,3,"\node{Ax:NP}") ;
```

```
  draw fromtopaths(followpath,p4,1,p4,2) ;
  draw fromtopaths(followpath,p5,1,p5,2) ;
  draw fromtopaths(followpath,p4,2,p4,3,"\node{P:VP}") ;
  draw fromtopaths(followpath,p6,1,p6,2) ;
  draw fromtopaths(followpath,p7,1,p7,2) ;
  draw fromtopaths(followpath,p6,2,p6,3,"\node{PCs:AdjP}") ;
  draw fromtopaths(followpath,p0,1,p0,2) ;
  draw fromtopaths(followpath,p1,3,p1,4) ;
  draw fromtopaths(followpath,p0,2,p0,3,"\node{S:NP}") ;
  draw fromtopaths(followpath,p4,3,p4,4) ;
  draw fromtopaths(followpath,p6,3,p6,4) ;
  draw fromtopaths(followpath,p4,4,p4,5,"\node{Pred:PredP}") ;
  draw node(p0,4.5,"\node{Cl}") ;
  draw fromtopaths(followpath,p0,3,p0,4.5) ;
  draw fromtopaths(followpath,p4,5,p4,6) ;
\stopMPcode
```

Note that diagrams such as those constructed here will each be significantly different, making the writing of a general mechanism rather complex. For example, one might need to construct a tree branching up rather than down, or to the right (or left), or even following an arbitrary path, such as a random walk. These can all be achieved individually in MetaPost without too much difficulty.

A 3D projection

Although MetaPost is a 2D drawing language, it can be easily extended to work in 3D. Several attempts have been made in the past ranging from simple to complicated. Here, we will take a simple approach.

The MetaPost language includes a triplet variable type, used to handle `rgb` colors (it also has a quadruplet type used for `cmyk` colors). We will use this `triplet` type to hold 3D coordinates. There is a separate ConTEXt module, entitled `three`, which creates a new MetaPost instance (also named `three`), which loads a set of macros that can be used to manipulate these triplet coordinates.

```
\usemodule [three]

\startMPcode{three}
  % code here
\stopMPcode
```

For our purposes here, only one function is really necessary: `projection()` that maps a 3D coordinate to a 2D projection on the page. This will not be a perspective projection having a viewpoint and a focus point, but rather a very simple oblique projection, useful for, e.g., pseudo-3D schematic drawings. The `Z` coordinate is taken to be `up` and the `Y` coordinate taken to be `right`, both in the page of the paper. The third coordinate `X` is an oblique projection in a right-hand coordinate system.

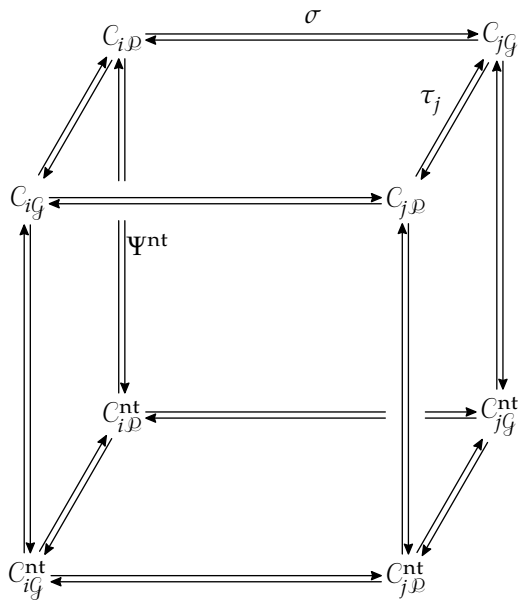

Figure 17.

Intended for schematic drawings, there is no automatic hidden-line removal nor effects like shading, and line crossings need to be handled manually (using `crossingunder` introduced previously). In Figure 17 we draw a simple cubical commutative diagram, with a node at each corner.

```
\startMPcode{three}
  clearnodepath ;
  nodepath = (projection Origin --
              projection (1,0,0) --
              projection (1,1,0) --
              projection (0,1,0) --
              projection (0,1,1) --
              projection (1,1,1) --
              projection (1,0,1) --
              projection (0,0,1) --
              cycle) scaled 5cm ;

  draw node(0,"\node{${\cal C}_{i\cal P}^{\mathrm{nt}}$") ;
  draw node(1,"\node{${\cal C}_{i\cal G}^{\mathrm{nt}}$") ;
  draw node(2,"\node{${\cal C}_{j\cal P}^{\mathrm{nt}}$") ;
  draw node(3,"\node{${\cal C}_{j\cal G}^{\mathrm{nt}}$") ;
  draw node(4,"\node{${\cal C}_{j\cal G}$") ;
  draw node(5,"\node{${\cal C}_{j\cal P}$") ;
  draw node(6,"\node{${\cal C}_{i\cal G}$") ;
  draw node(7,"\node{${\cal C}_{i\cal P}$") ;
```

```
  interim crossingscale := 30 ;
  drawdoublearrows fromto(0,0,1) ;
  drawdoublearrows fromto(0,1,2) ;
  drawdoublearrows fromto(0,2,3) ;
  drawdoublearrows fromto(0,3,0)
     crossingunder fromto(0,2,5) ;

  drawdoublearrows fromto(0,7,6) ;
  drawdoublearrows fromto(0,6,5) ;
  drawdoublearrows fromto.ulft(0,5,4,"\node{$_j$~}") ;
  drawdoublearrows fromto.top (0,7,4,"\node{$$}") ;

  drawdoublearrows fromto.lrt(0,0,7,"\node{$^{\mathrm{nt}}$}")
     crossingunder fromto(0,6,5) ;
  drawdoublearrows fromto(0,1,6) ;
  drawdoublearrows fromto(0,2,5) ;
  drawdoublearrows fromto(0,3,4) ;
\stopMPcode
```

Note the use of `drawdoublearrows`, a new MetaFun command that is introduced here.

Two final examples

We end this manual with two examples of more advanced commutative diagrams. The following example, shown in Figure 18, illustrates what in category theory is called a *pullback*. It is inspired from an example given in the TikZ CD (commutative diagrams) package.

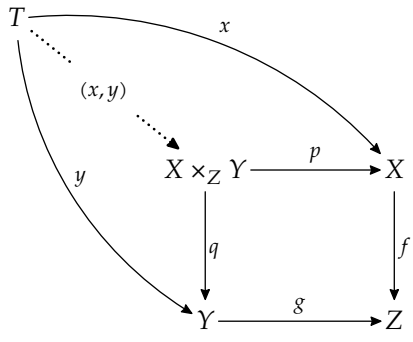

Figure 18.

The arrow labeled "(x,y)" is drawn `dashed withdots` and illustrates how the line gets broken, implicitly `crossingunder` its centered label.

```
\startnodes [dx=2.5cm,dy=2cm,alternative=arrow]
  \placenode [0, 0] {\node{$X\times_Z Y$}}
```

```
    \placenode [1, 0] {\node{$X$}}
    \placenode [1,-1] {\node{$Z$}}
    \placenode [0,-1] {\node{$Y$}}
    \placenode [-1,1] {\node{$T$}}

    \connectnodes [0,1] [position=top,   label={\smallnode{$p$}}]
    \connectnodes [1,2] [position=right, label={\smallnode{$f$}}]
    \connectnodes [0,3] [position=right, label={\smallnode{$q$}}]
    \connectnodes [3,2] [position=top,   label={\smallnode{$g$}}]
    \connectnodes [4,0] [option=dotted,rulethickness=1pt,
                                         label={\smallnode{$(x,y)$}}]
    \connectnodes [4,1] [offset=+.13,position=top,
                                         label={\smallnode{$x$}}]
    \connectnodes [4,3] [offset=-.13,position=topright,
                                         label={\smallnode{$y$}}]
  \stopnodes
```

The previous diagram was drawn using the ConTEXt interface. Our final example, shown in Figure 19, gives another "real-life" example of a categorical pullback, also inspired by TikZ-CD, but this time drawn through the MetaPost interface and solving for positions.

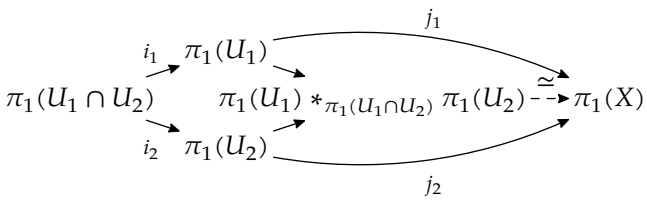

Figure 19.

```
  \startMPcode
    save nodepath ; save l ; l = 5ahlength ;
    save A, B, C, D, E ;
    pair A, B, C, D, E ;
    A.i = 0 ; A = makenode(A.i,"\node{$\pi_1(U_1\cap U_2)$}") ;
    B.i = 1 ; B = makenode(B.i,
    "\node{$\pi_1(U_1)\ast_{\pi_1(U_1\cap U_2)}\pi_1(U_2)$}") ;
    C.i = 2 ; C = makenode(C.i,"\node{$\pi_1(X)$}") ;
    D.i = 3 ; D = makenode(D.i,"\node{$\pi_1(U_2)$}") ;
    E.i = 4 ; E = makenode(E.i,"\node{$\pi_1(U_1)$}") ;
    A = origin ;
    B = A + betweennodes.rt(nodepath,A.i,nodepath,B.i) + (  l,0) ;
    C = B + betweennodes.rt(nodepath,B.i,nodepath,C.i) + (.7l,0) ;
    D = .5[A,B] + (0,-.9l) ;
    E = .5[A,B] + (0, .9l) ;
```

```
  for i = A.i, B.i, C.i, D.i, E.i :
      draw node(i) ;
  endfor
  drawarrow fromto.llft( 0,A.i,D.i,"\smallnode{$i_2$}") ;
  drawarrow fromto.ulft( 0,A.i,E.i,"\smallnode{$i_1$}") ;
  drawarrow fromto     ( 0,D.i,B.i) ;
  drawarrow fromto     ( 0,E.i,B.i) ;
  drawarrow fromto.urt( .1,E.i,C.i,"\smallnode{$j_1$}") ;
  drawarrow fromto.lrt(-.1,D.i,C.i,"\smallnode{$j_2$}") ;
  drawarrow fromto.top(  0,B.i,C.i) dashed evenly ;
  draw textext.top("{\tfxx\strut$\simeq$}")
      shifted point .4 of fromto(0,B.i,C.i) ;
\stopMPcode
```

Comparison with the `chart` module

The present `nodes` module can be used to draw flowcharts and relational-diagrams. It is useful to compare it to the ConTEXt `chart` module, expressly designed to produce process flowcharts.

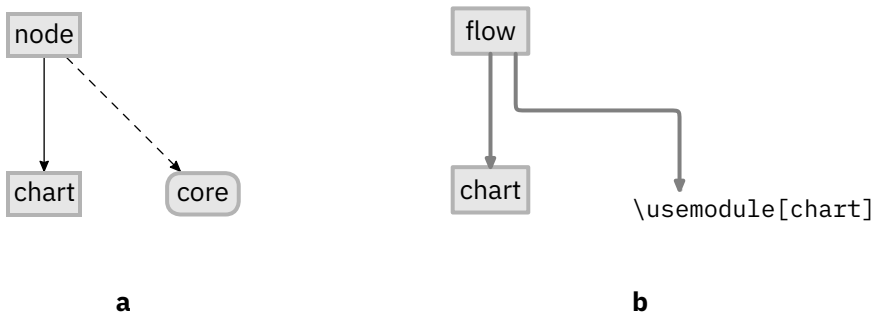

Figure 20. Compare (a) nodes and (b) chart.

Figure 20 shows a comparison between the results of the nodes and `chart` module.

```
\startnodes [dx=2.1cm,dy=-2.1cm,alternative=arrow]
    \placenode [1,1] [reference=A] {\cnode{node}}
    \placenode [1,2] [reference=B] {\cnode{chart}}
    \placenode [2,2] [reference=C] {\cnode[corner=round]{core}}
    \connectnodes [A,B]
    \connectnodes [A,C] [option=dashed]
\stopnodes
```

The `framed` instance `\cnode` was defined here to mimic the style of the `chart` shapes. The code producing Figure 20b is a bit more verbose (please refer to the chart module manual for more information):

```
\setupFLOWcharts
    [width=1cm,
     height=.6cm,
     dx=.75cm,
     dy=.75cm]

\startFLOWchart [flowchart]
    \startFLOWcell
        \location {1,1} \name {A}
        \shape {action} \text {flow}
        \connection [bt] {B}
        \connection [pbt] {C}
    \stopFLOWcell
    \startFLOWcell
        \location {1,2} \name {B}
        \shape {action} \text {chart}
    \stopFLOWcell
    \startFLOWcell
        \location {2,2} \name {C}
        \shape {node}
    \stopFLOWcell
\stopFLOWchart

\FLOWchart [flowchart]
```

The main difference, however, is that a flowchart is laid out on a worksheet of cells separated by space where the flowlines are located, as shown in Figure 21. A traditional drawing template and instructions for its use are shown in Figure 22.

The `node` core macros, on the other hand, are not constrained to cells on a regularly-spaced grid: as we have shown earlier, 'nodes' can be placed at any position, defining an arbitrary path. Furthermore, using the equation solving capability of MetaPost, nodes can be placed relative to one another. Whereas the ConTEXt interface, for convenience, defines a 2D grid (that may be oblique), coordinates on this grid need not be integer.

Flowcharts and entity-relational diagrams (often) use orthogonal connection lines. This is by design in the `chart` module and can be achieved in the `node` macros by adding (empty) nodes or by defining the node path containing these orthogonal points and instructing the connecting lines to follow these paths, when needed.

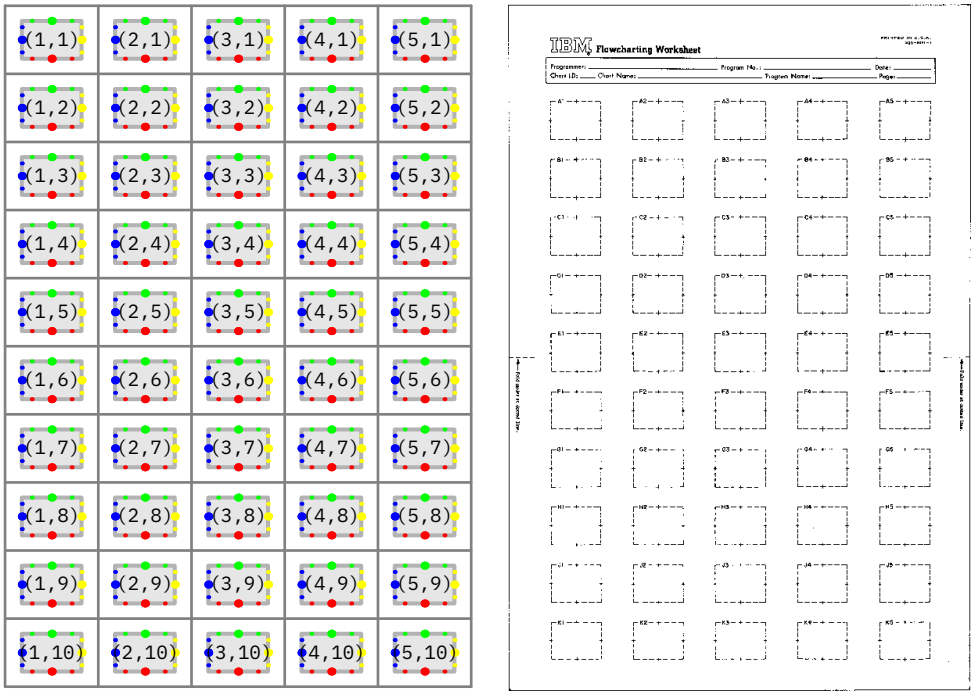

Figure 21. (a) ConTEXt chart module cells and (b) Flowcharting Worksheet.

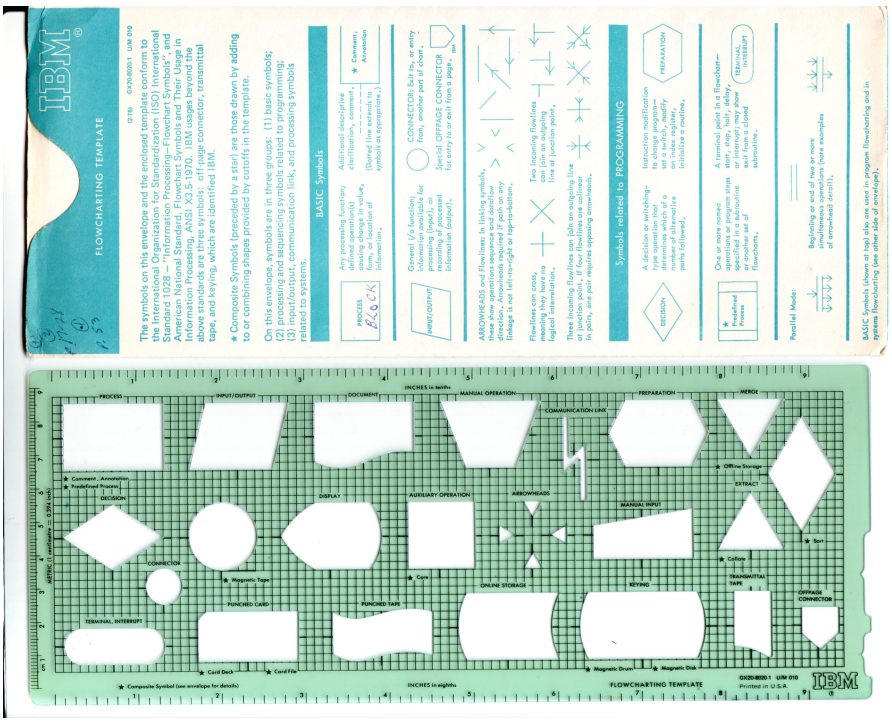

Figure 22. Flowcharting template

Conclusions

There was initial consensus at the 2017 ConTEXt Meeting in Maibach, Germany, where a version of this package was presented, that there was little use of developing a purely ConTEXt interface. Rather, the MetaPost package should be sufficiently accessible. Since then, however, we decided that the development of a derivative ConTEXt interface implementing some basic functionality could indeed be useful for many users, although it will necessarily remain somewhat limited. This was presented at the 2018 ConTEXt Meeting in Sibřina, Czech Republic[10] and documented here. Users are recommended to turn to the pure MetaPost interface when more sophisticated functionality is needed.

Acknowledgements

This module was inspired by a request made by Idris Samawi Hamid to draw a natural transformation diagram (Figure 4). The MetaPost macros that were developed then benefited from improvements suggested by Hans Hagen as well as inspiration provided by Taco Hoekwater.

References

Bethe, H. A. (1939). Energy Production in Stars. *Phys. Rev.*, *55*, 103–103. doi:10.1103/PhysRev.55.103; Bethe, H. A. (1939). Energy Production in Stars. *Phys. Rev.*, *55*, 434–456. doi:10.1103/PhysRev.55.434 (p. 67)

Braslau, A., Hamid, I. S., & Hagen, H. (2018). ConTeXt nodes: commutative diagrams and related graphics. *TUGboat*, *39*(1). Retrieved from https://www.tug.org/TUGboat/Contents/contents39-1.html (p. 57)

Krebs, H. A. (1946). Cyclic processes in living matter. *Enzymologia*, *12*, 88–100. (p. 65 and p. 67)

Lawvere, F. W. & Schanuel, S. H. (2009). *Conceptual Mathematics: A first introduction to categories*. (2nd ed.). Cambridge, UK: Cambridge University Press. (p. 57)

[10] We discussed at the 2018 ConTEXt Meeting application to entity-relationship diagrams, and the use of the 'crow's-foot' notation for logical relations, an 'arrow' extension to be considered for addition to MetaFun.