August 16, 2025

The Snail Tutorial

Li Yanrui (lyr.m2@live.cn)

Introduction

Maybe everyone knows MetaPost is great at drawing accurate science diagrams. But that's not the whole story! Think of it this way, it's actually a programming language for drawing vector graphics with simple syntax. You can write code to create the same great things as in those industry software (e.g., Adobe Illustrator, CorelDraw or Inkscape).

Don't believe the above claim I made. Let's be real. I've never worked as a graphic designer. Visual tools are much faster and easier for design work than coding with MetaPost. Exactly like how most pepole prefer movies over books these days for storytelling.

However, in this technological era, I wrote a compact MetaPost module for drawing flowchart and diagrammatic illustrations. Is this somehow a lamentable act? Certainly not. I have never intended to resist technological progress. Our era pursues its trajectory; I pursue mine. The primary driver for writing this module was the absence of satisfactory flowchart software on my Linux desktop.

This module is named snail. True to its name, it's slow at drawing diagrams.

Its development has been even slower. Around 2018, while properly learning MetaPost for the first time, I wrote the version 1 as practice work. It did not work well. In fact, after finishing it, I showed it off to a few friends then never used it again. In 2023, I relearned MetaPost and created the version 2. This one actually worked—I'd say it worked well. But I discovered MetaPost supports Chinese variable and macro names, so I built this version as a Chinese diagramming language. Both the variables and macros got quirky Chinese names. That's why I've never showed it to anyone since finishing it.

Now we've reached 2025. These past seven years? Honestly I've achieved nothing remarkable. Feeling low, I revisited MetaPost and wrote some documentation. This finally fulfilled my wish. Two years ago, after finishing ConTEXt notes[1], I'd wanted to write similar guides for MetaPost language. Along the way, third version of snail took shape. Maybe this is all I'm capable of, simple tools for simple needs.

[1] Please see https://github.com/liyanrui/ConTeXt-notes.

1 First Snail Diagram

This is the smallest possible snail drawing environment. All that's left is to write some MetaPost code inside the `MPpage` environment.

```
\usemodule[snail]
\startMPpage
% put metapost code here!
\stopMPpage
```

To compile a TEX source file foo.tex into foo.pdf, use

```
$ context foo.tex
```

or

```
$ context foo
```

Then you can get the foo.pdf file in the same directory. The above process can be expressed as the following snail code.

```
\usemodule[snail]
\startMPpage
snailfam_t a[];
slug(a1, "foo.tex");
snail(a2, "context") at (2.5cm, 0);
slug(a3, "foo.pdf") at (5cm, 0);
showsnails a1, a2, a3;

showflow a1 xto a2;
showflow a2 xto a3;
\stopMPpage
```

Example 1 Your first diagram

2 Nodes

All snail objects, i.e. nodes, must be declared before defining them. There are two declaration methods. The first one declares a group of nodes with the macro `snail_t`. The other one declares a group node sequence (or array) with the macro `snailfam_t`.

For example, to declare the nodes `foo` and `bar`, use

```
snail_t foo, bar;
```

To create multiple sequence of nodes, use MetaPost's array syntax. The following declares two node array `A` and `B`:

```
snailfam_t A, B;
```

The elements of a node array are accessible with indices, for example, `A[1]`, `A[2]`, ⋯ or `A1`, `A2`, ⋯.

Snail provides three node types constructed via the macros `snail`, `slug` and `avatar`. `snail` creates nodes with visiable frames, while `slug` creates frameless nodes. For specialized requirements, `avatar` import external figures as nodes. By default, the centers of newly created nodes are at the origin `(0, 0)`. The `at` macro can set a node's center when creating it. The `showsnails` macro draws all nodes that it accept.

The following example creates three nodes of different types.

```
snailfam_t a;
snail(a1, "I am $a_1$") at (4cm, 0);
slug(a2, "I am $a_2$") at (3cm, 1.5cm);
avatar(a3, "demo.png", 3cm, "auto");

showsnails a1, a2, a3;
```

I am a_2

I am a_1

Example 2 Three kinds of nodes

The second argument of `avatar` is the image file or path name. The third and fourth arguments are the width and height of the figure respectively. You can set just one of them; put `"auto"` for the other and the snail module will calculate it from the figure's aspect ratio.

3 Anchors

Every snail node has 25 anchor points. For the node `foo`, they are stored in `foo.anchors`, a two-dimensional array. The center of `foo` is `foo.anchors[0][0]` as shown below.

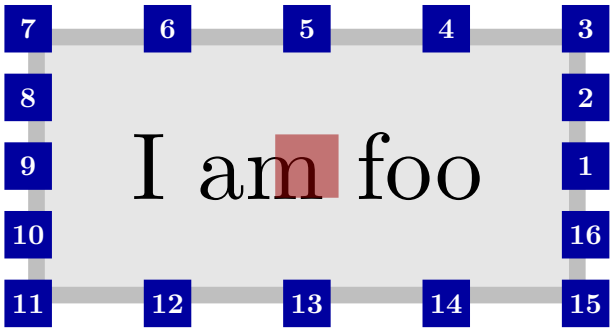
```
snail_t foo;
snail(foo, "I am foo");
showsnails foo;

% draw foo.anchors[0][0]
pickup pensquare scaled 8pt;
draw foo.anchors[0][0]
     withcolor transparent(1, .5, darkred);
```

I am foo

Example 3 Center anchor

If `foo.anchors[0][0]` is taken as the single anchor at level 0, then `foo.anchors[1][1]` through `foo.anchors[1][16]` are the 16 anchors at level 1. The following example lables the indices of these anchors at this level.

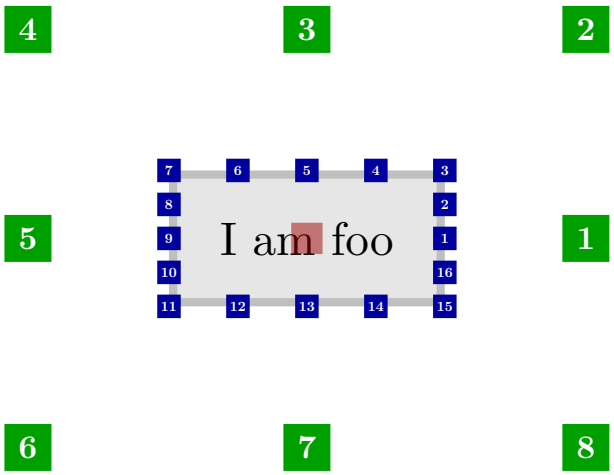
```
for i = 1 upto 16:
  draw foo.anchors[1][i] withcolor darkblue;
  draw textext(decimal i) shifted foo.anchors[1][i] withcolor white;
endfor;
```



Example 4 First anchor level

From `foo.anchors[2][1]` to `foo.anchors[2][8]` constitute the second level of the anchor points of the node `foo`. They lie outside the node's frame.

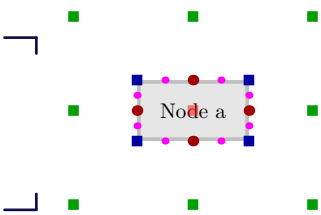
```
for i = 1 upto 8:
  draw foo.anchors[2][i] withcolor darkgray;
  draw textext(decimal i) shifted foo.anchors[2][i] withcolor white;
endfor;
```



Example 5 Second anchor level

To display all anchor points of a node, you can use the `snailenv.set` macro to set the snail module parameter `debug` to `true` before the `showsnails` macro statement.

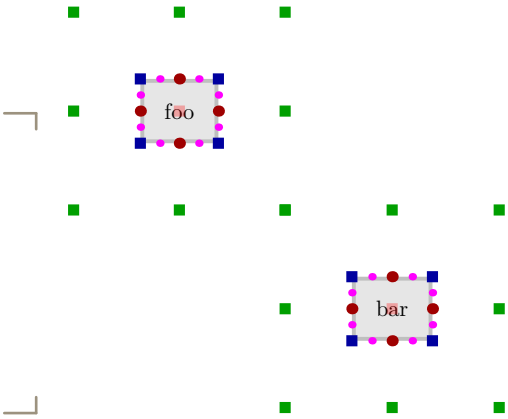
```
snail_t a;
snail(a, "Node a");
snailenv.set("debug", true);
showsnails a;
```



Example 6 Node debug mode

With help of anchor points, you can position a node by relative displacement with the snail syntax `put node at a from b`. Example 7 aligns the anchor point `bar.anchors[2][4]` with the anchor point `foo.anchors[2][8]`.

```
snail_t foo, bar;
snail(foo, "foo");
snail(bar, "bar");
put bar at foo.anchors[2][8] from bar.anchors[2][4];

snailenv.set("debug", true);
showsnails foo, bar;
```



Example 7 Node relative displacement

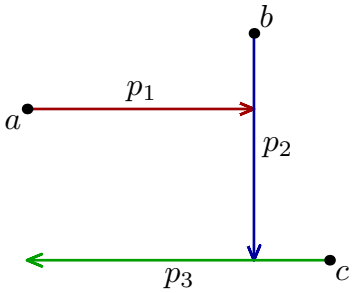
4 Regular Paths

The snail module provides a set of macros that construct paths composed solely of horizontal and vertical segments; such paths are called regular paths.

The simplest regular paths are built with the `xto` or `yto` macros. Example 8 creates a horizontal path `p` and a vertical path `q` using `xto` and `yto` repectively. The syntax `a xto b` is equivalent to `a -- (xpart b, ypart a)`, and `a yto b` is equivalent to `a -- (xpart a, ypart b)`.

Note that `xto` and `yto` can be applied directly to nodes; the resulting paths are called flows. A flow can be displayed with the `showflow` macro. Example 1 already demonstrates how to create and display flows.

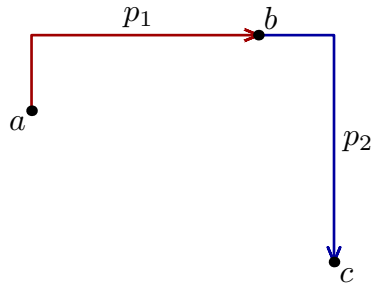
```
pair a, b, c;
a := (1cm, 2cm); b := (4cm, 3cm); c := (5cm, 0);

path p[];
p1 := a xto b; p2 := b yto c; p3 := c xto a;
drawarrow p1; drawarrow p2; drawarrow p3;
```



Example 8 Simplest regular paths

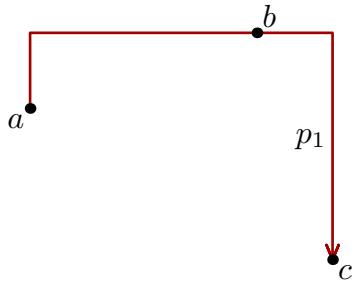
The `xyto` and `yxto` macros create regular paths with a single turn. The `xyto` macro indicates the single turn is from horizontal to vertical direction; `yxto` does the reverse. Their usage is shown in Example 9. These two macros can be combined to build more complex paths; see Example 10.

```
p1 := a yxto b;
p2 := b xyto c;
```



Example 9 Regular paths with single turn

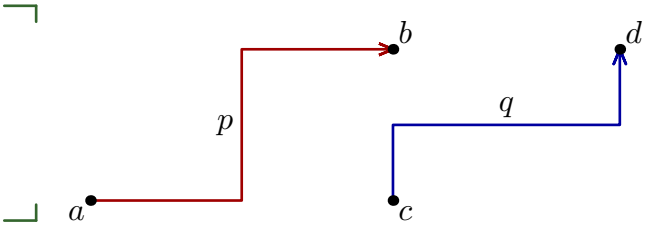
```
p1 := a yxto b xyto c;
```



Example 10 Regular paths with single turn

If you want a path with two single turns, you can use the `xyxto` and `yxyto` macros. `xyx` indicates turns from horizontal to vertical and back to horizontal. `yxy` indicates turns from vertical to horizontal and back to vertical direction. Their usage is shown in Example 11. These two macros can also be combined to build more complex paths.

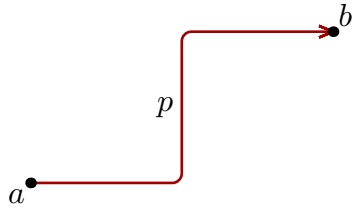
```
pair a, b, c, d;
a := (0, 0);      b := (4cm, 2cm);
c := (4cm, 0);    d := (7cm, 2cm);
path p, q;
p := a xyxto b;   q := c yxyto d;
```



Example 11 Regular paths with two turns

The regular paths produced by the `xyto`, `yxto`, `xyxto` and `yxyto` macros can be smoothed with the `road` macro; all turns in paths are rendered as fillets. The `road` macro takes two arguments: the first is a path variable, and the second is a regular path. After smoothing, the resulting path is assigned to the first argument.

```
pair a, b;
a := (0, 0); b := (4cm, 3cm);
path p;
road(p, a xyxto b);
```



Example 12 Smoothed paths

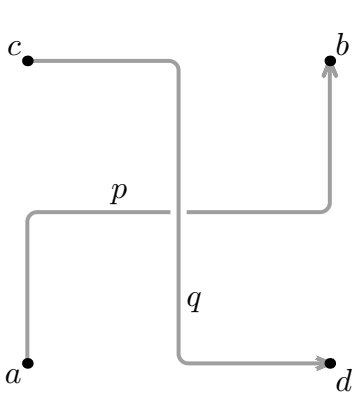
5 Path Crossing

Regular paths and the smoothed paths produced by `road`, are all ordinary path objects. Because they carry no special metadata, they can only be drawn one by one with Meta-

Post's `drawarrow` macro. Consequently, when paths intersect, it is difficult to add crossing indicators.

The snail module provides the macro `showroads`, which draws all paths as arguments in a single pass. Internally, it detects any intersections and renders them with short gaps in the lines to indicate that one path passes over another.

```
pair a, b, c, d;
a := (0,   0);  b := (4cm, 4cm);
c := (0, 4cm);  d := (4cm,   0);

path p, q;
road(p, a yxyto b);
road(q, c xyxto d);

showroads p, q;
```

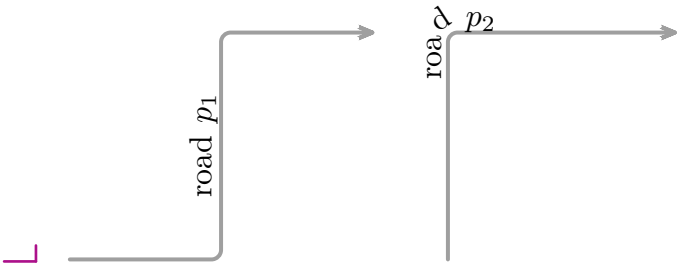


Example 13 Crossing paths

6 Path Annotation

The snail module provides the `annotate` macro for labeling paths. This macro places some text based on the middle point of a path. Example **14** gives the usage of `annotate` and its side effect.

```
path p[];
road(p1, (0, 0) xyxto (4cm, 3cm));
road(p2, (5cm, 0) yxto (8cm, 3cm));
showroads p1, p2;

annotate(p1, "road $p_1$", 1);
annotate(p2, "road $p_2$", 1);
```

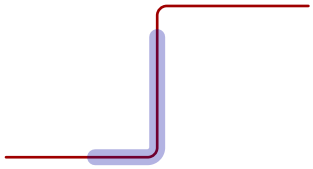


Example 14 Path annotations

The label of `p2` appears distorted because its center is aligned with the bend in `p2`. If you want to correct it, you need to use the `section` macro which can take a part of a path as the labeling target instead of the origin path.

In MetaPost language, a path is parametrized over the interval $[0, 1]$, so every point ont it is obtained by evaluating the path at some `t` in $[0, 1]$. The `section` marcro exploits this to extract a desired section from a path. Example **15** shows the part `[0.2, 0.6]` of the path `p`. By taking a section of the path as the labeling target, we can place a label at any position along this path.

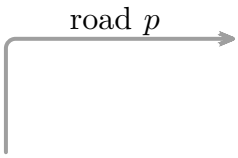
```
path p, q;
road(p, (0, 0) xyxto (4cm, 2cm));
q := section(p, .2, .6);

pickup pencircle scaled 1pt;
draw p withcolor darkred;
draw q withpen pencircle scaled 6pt
      withcolor transparent(1, .3, darkblue);
```



Example 15 Path section

Example **16** demonstrates how to place a label at a section of a path.

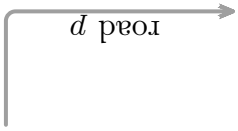
```
path p;
road(p, (0, 0) yxto (3cm, 1.5cm));
showroads p;
annotate(section(p, .3, 1), "road $p$", 1);
```



Example 16 Placing label on section

The third argument of the `annotate` macro specifies the orientation of the label text: a positive value aligns the text with the path's direction, while a negative value reverses it.

```
path p;
road(p, (0, 0) yxto (3cm, 1.5cm));
showroads p;
annotate(section(p, .3, 1), "road $p$", -1);
```

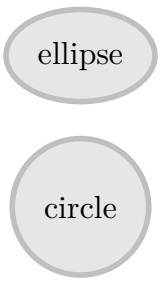


Example 17 Rerverse label

7 Irregular Nodes

You can cretate your own shapes to serve as the node's frame by setting the module parameter `frame.shape`. For example, if you need a ellipse node, you only set `frame.shape` to `fullcircle`. If you want a circle node, you need to set `frame.isotropic` to `true`.

```
snail_t foo, bar;

snailenv.set("frame.shape", "'fullcircle'");
snail(foo, "ellipse");

snailenv.set("frame.isotropic", true);
snail(bar, "circle") at (0, -2cm);
```

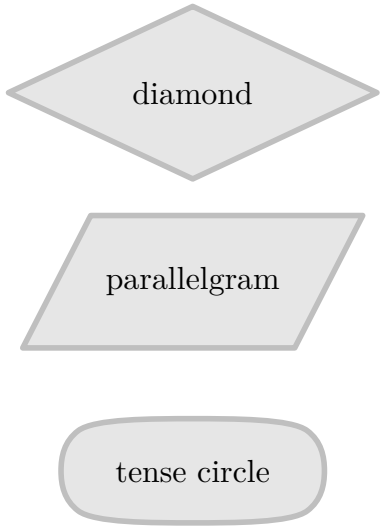


Example 18 Ellipse node

Example **19** applies the same technique to generate more irregular node types. The `fulldiamond` and `tensecircle` macros are defined in MetaFun. The path `datum` is

scaled and sheared fullsquare object. Note the path assigned to the module parameter `frame.shpae` must be enclosed in single quotation marks, so that the result is wrapped into a MetaPost's string object.

```
snailfam_t a;

snailenv.set("frame.shape", "'fulldiamond'");
snail(a1, "diamond");

path datum;
datum := fullsquare xyscaled (1, .25) slanted 1;
snailenv.set("frame.shape", "'datum'");
snail(a2, "parallelgram") at (0, -2.5cm);

snailenv.set("frame.shape", "'tensecircle(2, 1, .15)'");
snail(a3, "tense circle") at (0, -5cm);
```



Example 19 More irregular nodes

In fact, node's frame can be any closed path. In short, you can turn virtually any shape into a node, letting you draw diagrams that match your exact needs. After using irregular nodes, you can revert to snail's default node style with the `snailshapereset` macro.

```
snailshapereset;
```

8 Module Parameters

By now you've seen several examples where the `snailenv.set` macro is used to tweak the module parameters and thereby alter the diagram style. For example, to set the stroke thickness of a path to `1pt`, assign the parameter `path.thickness` as follows:

```
snailenv.set("path.thickness", 1pt);
```

If you want to change the default color of the path to `darkred`, simply

```
snailenv.set("path.color", darkred);
```

Note that the first argument to `snailenv.set` is always a string denoting the name of a module parameter, while the second argument may be any value or variable allowed in MetaPos except those of type `path` or `picture`.

Among the parameters, only `frame.shape` is of MetaPost's `path` type. To modify this parameter, refer to the previous examples: to pass such variables to the parameter table, you must enclosed them in single quotes then as MetaPost's string objects.

Some frequently-used parameters of the snail module are listed in the table below. This table is a Lua table because, within the ConTEXt environment, MetaPost and Lua have

become practically inseparable. While drawing, experiment freely—use `snailenv.set` to push the parameters in the table to exaggerated extremes and watch what each one actually does.

```
snailenv = {
   text = {
      fontsize = 'BodyFontSize', color = 'black', offset = '.125BodyFontSize'
   },
   frame = {
      shape = 'fullsquare', background = {color = '.9white'}, offset = 'BodyFontSize',
      thickness = '.175BodyFontSize', color = '.75white', isotropic = 'false',
      margin = '3BodyFontSize',
   },
   path = {
      thickness = '.125BodyFontSize', color = 'darkgray', directed = 'true',
      smooth = 'true', labeloffset = '.5BodyFontSize', fillet = '.3BodyFontSize',
      ahvariant = 1, ahdimple = 1, ahlength = '.5BodyFontSize',
   },
   debug = 'false'
}
```

Afterword

Everything the snail module can do has now been laid out. The tour was intentionally made of tiny vignettes; I never produced one genuinely useful flowchart—such a grand scene is meant for you.

If you actually plan to draw with snail, start on paper. A rough sketch is enough: mark the nodes and their approximate positions. From that sketch, create the nodes and place them, then switch the module's debug parameter to true so every node and its anchors are drawn. Once the nodes and anchors are on the page, building and annotating the paths becomes nothing more than a game of Snake.