Paths, pairs, pens and transforms

Abstract

This article tries to explain everything related to paths, pairs, pens and transforms in MetaPost. A fair bit of familiarity with MetaPost's data types and general syntax is assumed. In particular, I assume you have read the 'sparks, tags, suffixes and subscripts' article.

I will first discuss the creating of paths, followed by the creating of pairs, and then the creating of pens. Finally, I will discuss the operations on those items, for instance, by using transformations.

Defining a path

In MetaPost, `path`s are the data structures that are added to pictures in order to create visible shapes in the output.

At its core, a `known` path is a list of data objects describing Bézier curve segments. These objects are called 'knots', and they are constructed internally from a list of `pair`s, other embedded (sub)paths, and the user-specified options on how to connect them.

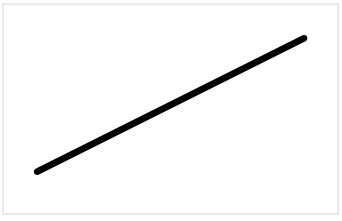
To get it out of the way, here is the syntax definition for path expressions:

⟨path primary⟩ → ⟨pair primary⟩ | ⟨path variable⟩ | ⟨path argument⟩ | `(`⟨path expression⟩`)`
| `begingroup` ⟨statement list⟩⟨path expression⟩ `endgroup`
| `makepath` ⟨pen primary⟩ | `makepath` ⟨future pen primary⟩
| `reverse` ⟨path primary⟩
| `subpath` ⟨pair expression⟩ `of` ⟨path primary⟩
| `envelope` ⟨pen primary⟩ `of` ⟨path primary⟩
⟨path secondary⟩ → ⟨pair secondary⟩ | ⟨path primary⟩
| ⟨path secondary⟩⟨transformer⟩
⟨path tertiary⟩ → ⟨pair tertiary⟩ | ⟨path secondary⟩
⟨path expression⟩ → ⟨pair expression⟩ | ⟨path tertiary⟩
| ⟨path subexpression⟩⟨direction specifier⟩
| ⟨path subexpression⟩⟨path join⟩ `cycle`
⟨path subexpression⟩ → ⟨path expression not ending with direction specifier⟩
| ⟨path subexpression⟩⟨path join⟩⟨path tertiary⟩
⟨path join⟩ → ⟨direction specifier⟩⟨basic path join⟩⟨direction specifier⟩
⟨direction specifier⟩ → ⟨empty⟩
| `{ curl` ⟨numeric expression⟩ `}`
| `{` ⟨pair expression⟩ `}`
| `{` ⟨numeric expression⟩ `,` ⟨numeric expression⟩ `}`
⟨basic path join⟩ → `&` | `..`
| `..`⟨tension⟩`..`
| `..`⟨controls⟩`..`
⟨tension⟩ → `tension` ⟨tension amount⟩
| `tension` ⟨tension amount⟩ `and` ⟨tension amount⟩
⟨tension amount⟩ → ⟨numeric primary⟩
| `atleast` ⟨numeric primary⟩
⟨controls⟩ → `controls`⟨pair primary⟩
| `controls` ⟨pair primary⟩ `and` ⟨pair primary⟩

This is the formal definition from the Metafont book, and as formal definitions go it is a bit awkward. Do not stare at it for too long, because the reality of specifying paths is fairly straightforward and natural. Let's instead just look at a few simple examples.

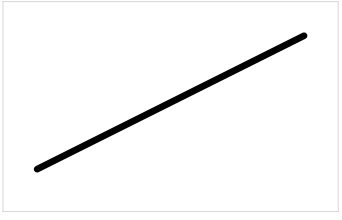
The simplest case is just from a few pair primaries and a basic path join:

```
path p;
p = (0,0)..(200,100);
draw p;
```



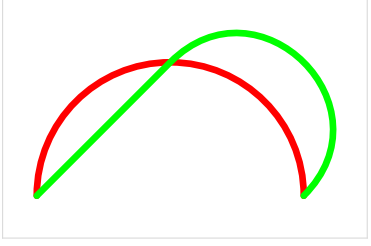
Nearly as easy is to form a path from a path (sub)expression:

```
path p;
p = ((0,0)..(200,100));
draw p;
```



This part of the syntax is about adding `()` grouping, just like in calculus. It makes sure that the expression inside the parentheses is converted into a path first, before any of the outer processing happens. This can be useful because in some cases subsequent `path join`s can have an effect on prior bits of the path. Here is an example of both:

```
path p,q;
p = (0,0)..(100,100)..(200,0);
q = ((0,0)..(100,100))..(200,0);
draw p withcolor red;
draw q withcolor green;
```



Of course, pairs can also be specified using variable names (we will look at the formal syntax of pairs later):

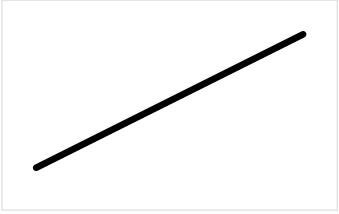
```
path p;
pair startp, endp;
startp = (0,0);
endp = (200,100);
p = startp..endp;
draw p;
```

Or even from a single point:

```
path p;
pair startp;
startp = (0,0);
p = startp;
draw p;
```

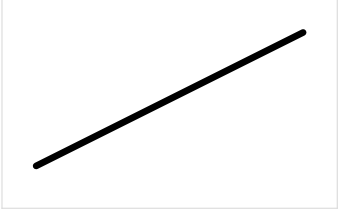
Or from another path:

```
path p, q;
q = (0,0)..(200,100);
p = q;
draw p;
```



Or from the `reverse` of another path:

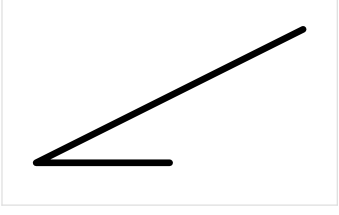
```
path p, q;
q = (0,0)..(200,100);
p = reverse q;
draw p;
```



While there is no visual difference in the above image, as a result of the `reverse` operator the beginning and end of the path `q` have been flipped, as can be clearly seen from the following examples:

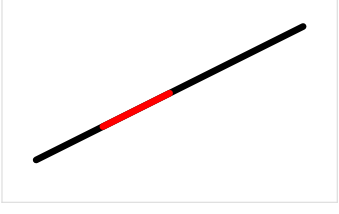
```
path p, q;
q = (0,0)..(200,100);
p = q--(100,0);
draw p;
```

```
path p, q;
q = (0,0)..(200,100);
p = reverse q--(100,0);
draw p;
```



It is also possible to create a new path from just a section of another path:

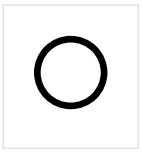
```
path p, q;
q = (0,0)..(200,100);
p = subpath (0.25,0.5) of q;
draw q;
draw p withcolor red;
```



The two numbers that form the `pair expression` argument to `subpath` signify a section of the 'travel' along the given `path primary` following the `of` keyword. We will revisit `subpath` and path lengths later.

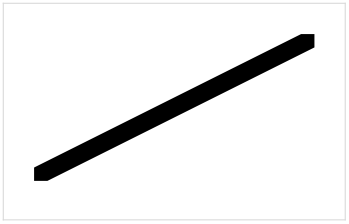
And from a pen:

```
path p;
p = makepath pencircle scaled 50;
draw p;
```



And finally, from the outline of a path drawn with a pen. We will talk about pens in a following section, but a bit of a sneak peak is needed to wrap up the path construction options. If this current bit is confusing, just skip ahead for now and come back to it later:

```
path p,q;
q = (0,0)..(200,100);
p = envelope pensquare scaled 10 of q;
fill p;
```



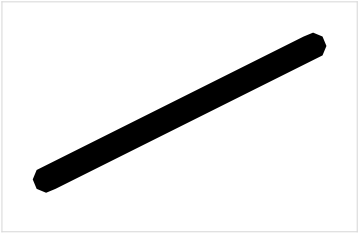
This last option for path construction comes with a few warnings and hints:

□ For envelope to work properly in the current version of MetaPost, the pen needs to be polygonal. Elliptical pens, like the built-in pencircle, will not work.

Here are two workarounds, both of which make use of primitives that have not been covered yet. However, the function of these primitives should hopefully be clear from the examples.

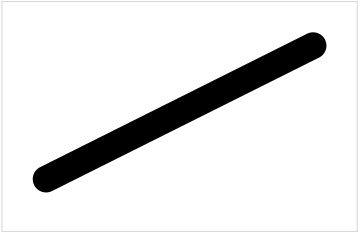
First, if actual precision in the curves is not very important, you can get away with converting the pencircle to a path, then converting that path back to a pen. This procedure creates a polygonal pen with eight sides:

```
path p,q;
pen trick;
q = (0,0)..(200,100);
trick = makepen makepath pencircle;
p = envelope trick scaled 20 of q;
fill p;
```



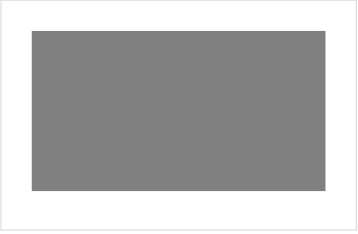
Or second, if greater curve precision is indeed needed, you can create a pen with a larger number of vertices by using a for loop to construct a new path from the pencircle:

```
path p,q,r,s;
pen trick;
q = (0,0)..(200,100);
s = makepath pencircle;
r = for i = 0 step 0.1 until 8:
    (point i of s) -- endfor cycle;
trick = makepen r;
p = envelope trick scaled 20 of q;
fill p;
```

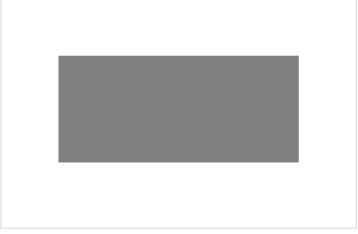


□ To get the 'other' side of a cyclic path, `reverse` the path:

```
path p,q;
q = (0,0)--(200,0)--(200,100)--
    (0,100)--cycle;
p = envelope pensquare scaled 20
    of q;
fill p withcolor 0.5;
```



```
path p,q;
q = (0,0)--(200,0)--(200,100)--
    (0,100)--cycle;
p = envelope pensquare scaled 20
    of reverse q;
fill p withcolor 0.5;
```

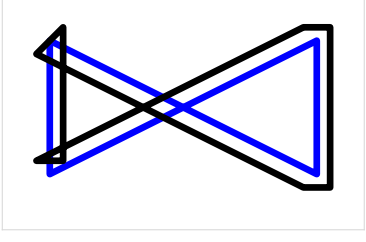


□ The path created by `envelope` is not always completely 'clean' or even 'correct' in the current version of MetaPost; it is intended to produce the visual envelope, not the 'best' path to create that envelope. It also still has bugs, especially with self-intersecting paths.

For instance, it may contain self-intersections or superfluous points. For that reason, it is mostly useful with `fill` instead of `draw`, and only with quite simple paths.

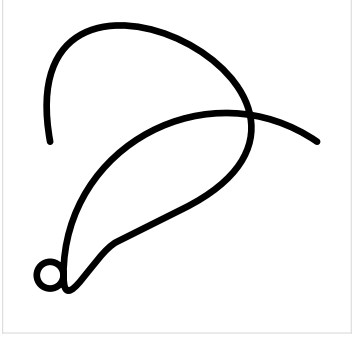
You can see some of the problems that may occur in the next example:

```
path p,q;
q = (0,0)--(200,100)--(200,0)--
    (0,100)--cycle;
draw q withcolor blue;
p = envelope pensquare scaled 20
    of reverse q;
draw p;
```



To wrap up this demonstration of how to create a path, here is a combination of almost all those options:

```
path p, q;
pair endp;
endp = (200,100);
q = (0,0)..(200,100);
p = (0,100) ..
    reverse (subpath (0.25,0.5) of q) ..
    makepath pencircle scaled 20 ..
    endp;
draw p;
```



Path directions and connectors

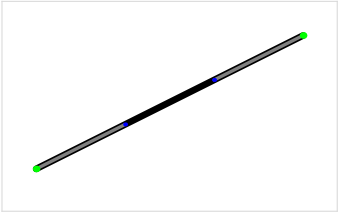
This section is about how to control the shape of a path: how the points are connected to form actual curves. Each point will be connected to the next point by a Bézier curve segment that is controlled by the two points themselves, and two extra 'control points'.

To help with visualization, the examples in this section use a macro from MetaFun called `detaileddraw` (with some adjusted preset options) to show the actual points and control points in the example paths. With normal `draw` it would just show black lines as in the previous examples.

Direction specifiers

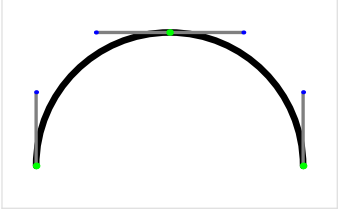
We will start by talking about direction specifiers. In the simplest case, the example below shows an empty direction specifier. Here two pairs are joined just by the two dots that we frequently saw in the previous section.

```
path p;
p = (0,0)..(200,100);
detaileddraw p;
```



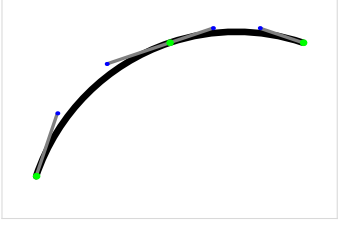
In this example, we are letting MetaPost decide by itself how it wants to connect the pairs. If there are only two pairs, the result is a straight line connecting them. If there are more than two more pairs in the sequence that are not in a straight line, MetaPost will try to create a nice curve connecting them:

```
path p;
p = (0,0)..(100,100)..(200,00);
detaileddraw p;
```



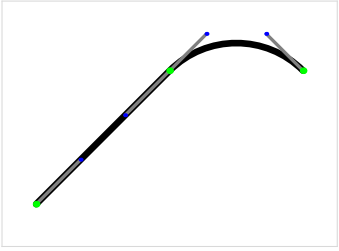
When left to its own devices, MetaPost will try to connect the points in such a way that the overall direction of the combined curve along the path only changes in a fluid way. The curvature at the start- and endpoints will attempt to follow a circular arc through the initial set of pairs. This can affect the directions quite a lot, of course:

```
path p;
p = (0,0)..(100,100)..(200,100);
detaileddraw p;
```



Just a reminder: parenthesis grouping has an effect on this logic, because it converts the path inside the parentheses into what is essentially a separate temporary nameless path. For example:

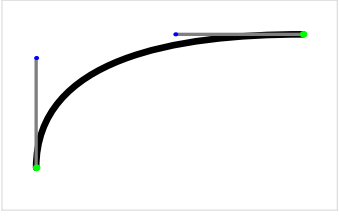
```
path p;
p = ((0,0)..(100,100))..(200,100);
detaileddraw p;
```



Also be aware that `tension` specifiers elsewhere in the path can have effect on the chosen directions. Later on we will see some examples of this.

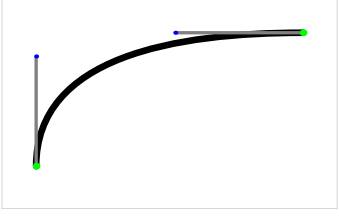
Direction vectors If you want to have explicit direction control, you can add a direction specifier, for example, using pair expressions that represent direction vectors:

```
path p;
p = (0,0){(0,1)}..{(1,0)}(200,100);
detaileddraw p;
```



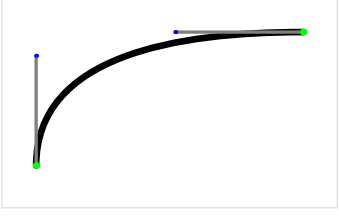
The previous example is not very readable; it is easier to understand using pair expressions stored in variables:

```
path p;
pair up,right;
up = (0,1);
right = (1,0);
p = (0,0){up}..{right}(200,100);
detaileddraw p;
```



Here the `(0,1)` definition of `up` means that this represents the unit-vector that moves 0 in the x direction and $+1$ in the y direction, i.e.: upwards. It could as easily have been defined as `(0,10)` and it would have made no difference, because the pair is interpreted as a direction vector which is then treated strictly as an angle. Adding a bigger y displacement has no effect as long as the x displacement remains zero, because only the direction of the vector is taken into account:

```
path p;
pair up,right;
up = (0,10);
right = (10,0);
p = (0,0){up}..{right}(200,100);
detaileddraw p;
```

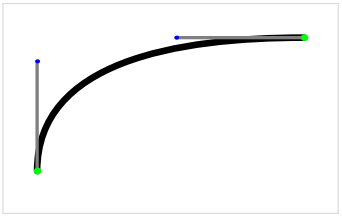


It should be obvious that the vector has to be completely known. If not, an error will be generated. On the other hand, a vector of `(0,0)` is acceptable. This has the same effect as not specifying a vector at all.

Side note: the pairs `up` and `right` (as well as `down` and `left`) are normally predefined by the macro package you are using, so the preceding two examples could each be three lines shorter.

In MetaPost macro packages there is usually also a macro defined called `dir`, which works like this:

```
path p;
p = (0,0){dir 90}..{dir 0}(200,100);
detaileddraw p;
```



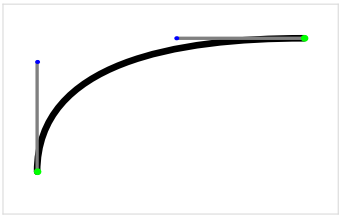
The actual definition of `dir` could be:

```
vardef dir primary d =
  right rotated d
enddef;
```

with `right` pair variable predefined as before.

The next direction specification option is nearly the same as the previous one. Apart from using a `pair expression`, it is also OK to use two known separate `numeric expression`s.

```
path p;
p = (0,0){0,1}..{1,0}(200,100);
detaileddraw p;
```

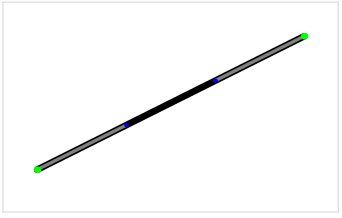


When building paths inside macros, either one or the other of these cases (`pair` or `numeric`) may be easier to work with. The end results are identical; there is always an internal vector constructed.

Curl specifiers The more advanced option has been kept for last; on start- and endpoint, you can instead use a `curl` specification.

First off, here is an example of the syntax:

```
path p;
p = (0,0){curl 0}..(200,100);
detaileddraw p;
```



The resulting image does not explain a lot, but that is because the rules for `curl` are quite specific.

A curl specification is a number from 0 to infinity.

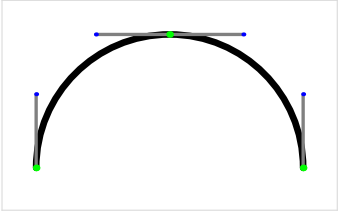
- ☐ It sets the amount of curliness (angle) at that point.
- ☐ If the requested amount of curl is high, it will adjust the curliness at adjacent points as well.
- ☐ Its assumed default value at ending points is 1.
- ☐ An explicit `curl` setting makes that point an ‘endpoint’ (a.k.a. a corner).

The vector-based method of specifying the directions from the earlier examples is always an absolute angle where `curl` is a relative approach that takes the rest of the path into account.

If there is no user-supplied direction specifier, `curl 1` is implied – which is exactly how MetaPost comes up with its default near-circular curves.

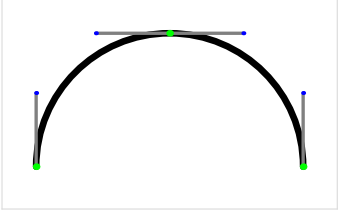
So:

```
path p;
p = (0,0)..(100,100)..(200,00);
detaileddraw p;
```



has the same effect as:

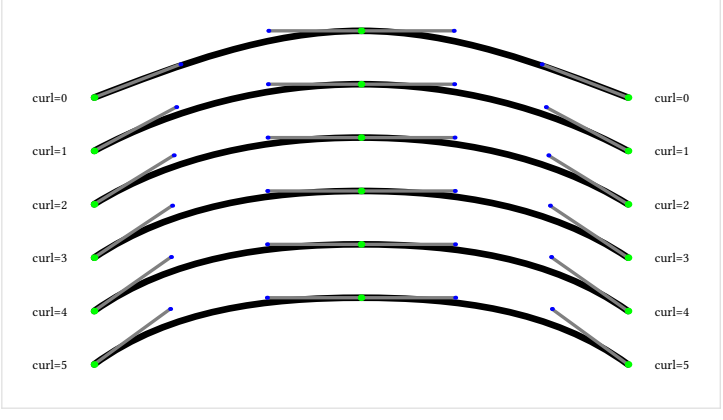
```
path p;
p = (0,0){curl 1}..
    (100,100)..
    {curl 1}(200,00);
detaileddraw p;
```



For a better understanding, it is easier to start by showing a progression. The next output was created using this example:

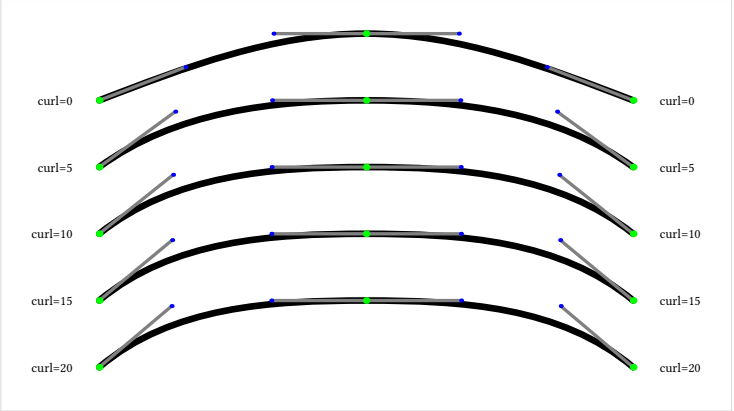
```
path p;
yoff := 40;
for d = 0 step 1 until 5:
   p := (0,0){curl d}..(200,50)..{curl d}(400,0);
   detaileddraw (p shifted (0,-yoff*d));
   draw textext.lft("curl=" & decimal d) shifted (-20,-yoff*d);
   draw textext.rt("curl=" & decimal d) shifted (420,-yoff*d);
endfor
```

There is some extra stuff there to print the labels, but the main thing the example does is create a set of paths with increasing settings for `curl` at both of the endpoints:

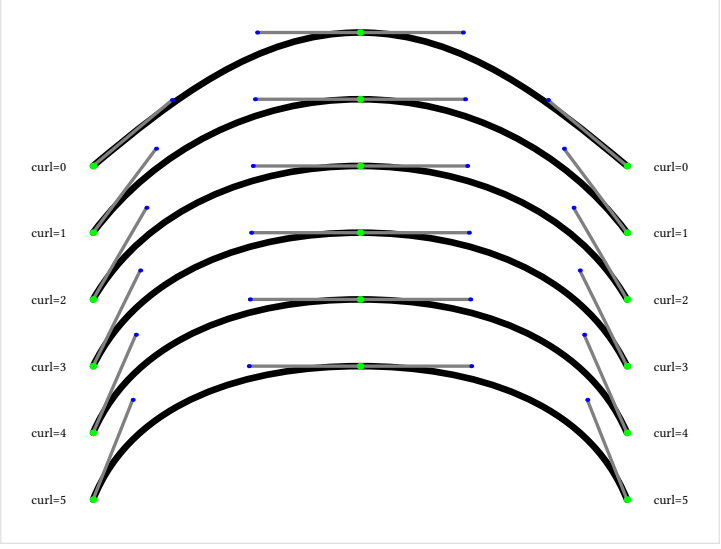


As you see, the angle (to the control point) increases when the `curl` value rises. But there is an upper limit above which it does not matter any more how much higher

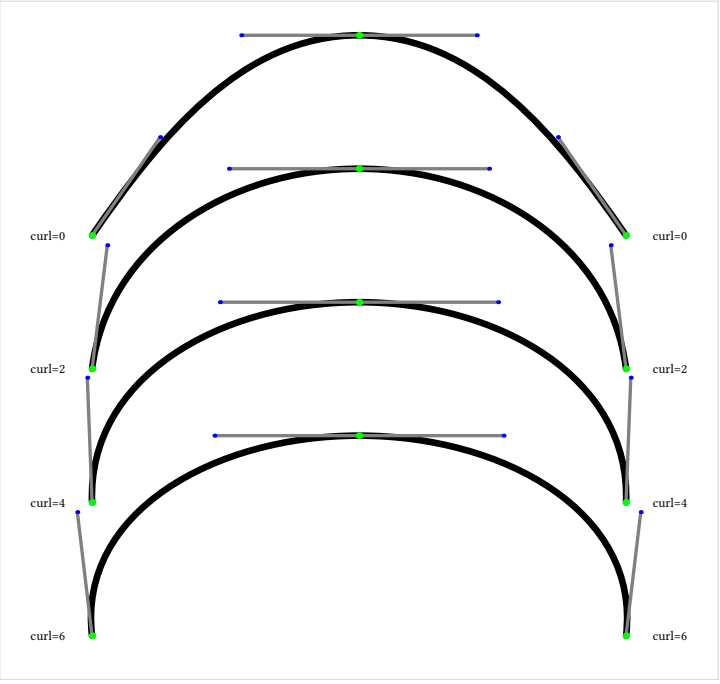
the `curl` value gets. Above a certain value, there is not enough curvature left for `curl` to be able to redistribute towards the endpoints:



While `curl` modifies the curvature of part of a curve segment, it is itself influenced by the length and required turning angle of that curve as well. Moving the middle point up to `(200,100)` while keeping everything else the same produces quite a different effect on the control points of the curve, because the required turning angle changes quite dramatically:



By moving the middle point up even higher to `(200,150)`, eventually the control points are pushed way off to the side:

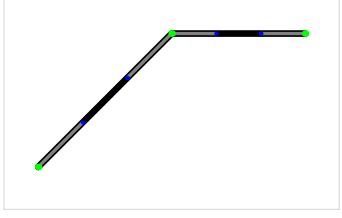


While the `curl` specifier takes some getting used to, it is a good tool to control the initial and final curves of a path since using a direction vector is not always straightforward, especially so if paths are rotated or otherwise transformed. On the downside, it may seem a bit temperamental, because the end result depends on other properties of the path in a way that is not easily predictable until one gains some experience with `curl`.

Side note: the fact that an explicit `curl` specification forces a point to behave as an 'endpoint', is what makes this definition work:

```
def -- = {curl 1}..{curl 1} enddef;
```

```
path p;
p = (0,0)--(100,100)--(200,100);
detaileddraw p;
```



The path equation with the `--` operator expands into:

```
p = (0,0){curl 1}..{curl 1}(100,100){curl 1}..{curl 1}(200,100);
```

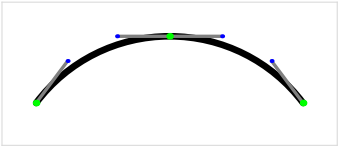
which makes every point a corner. The segments between those points is then filled using the '2 point' segment logic, resulting in straight lines.

Some final remarks about direction specifiers that are handy to know:

- ☐ explicit vectors are expressions, so you can do calculations while constructing the path.
- ☐ explicit incoming or outgoing `curl` and vector-based direction specifications migrate to the inverse side as well, if they are left empty.

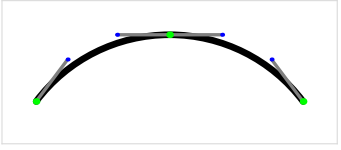
In an example:

```
path p;
p = (0,0)..{right}(100,50)..
    (200,0);
detaileddraw p;
```



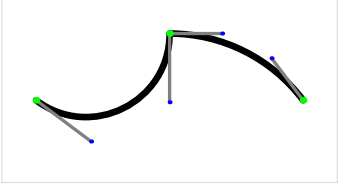
is the same as:

```
path p;
p = (0,0)..(100,50){right}..
    (200,0);
detaileddraw p;
```



□ while vectors do not force a point to behave as an 'endpoint', they can be used to create such, by specifying different values on the left and right side:

```
path p;
p = (0,0)..{up}(100,50){right}..
    (200,0);
detaileddraw p;
```

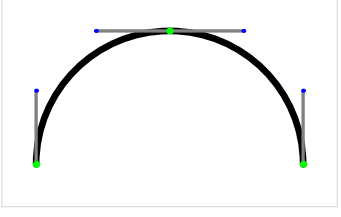


Path connectors

Now, let's talk about connectors. Connectors and directions together decide on the locations of the control points of the curve segments.

We have seen the simplest case a number of times already; it is just two consecutive dots:

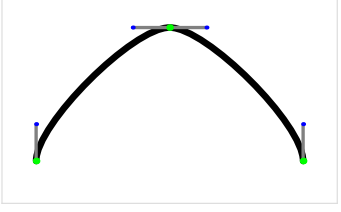
```
path p;
p = (0,0)..(100,100)..(200,0);
detaileddraw p;
```



Tension specifiers Internally, MetaPost has the concept of `tension`.

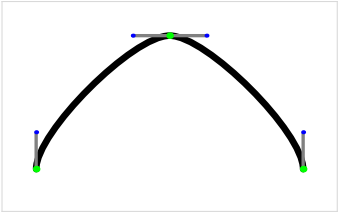
Amongst other things, the tension settings control how 'tight' the path is between two points:

```
path p;
p = (0,0)..tension 2 ..
    (100,100)..tension 2 ..
    (200,0);
detaileddraw p;
```



The example above used only one value, but actually there are two, one for each side of the segment:

```
path p;
p = (0,0)..tension 2 and 2..
    (100,100)..tension 2 and 2..
    (200,0);
detaileddraw p;
```



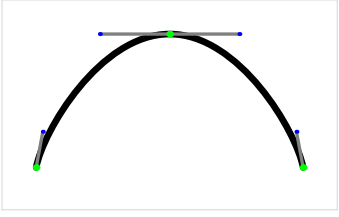
If you say just `tension 2`, it is silently interpreted as `tension 2 and 2`.

MetaPost's default for each segment where you do not set up an explicit `tension` is to assume that both values for path tension are set to 1, but the actual curvature and directions of the path can alter the effective values of the tensions to something more or less than one.

How `tension` controls the path segments exactly is quite technical, but it is important to note that the `tension` values can control the direction at points (if they were not set up by the user explicitly, of course).

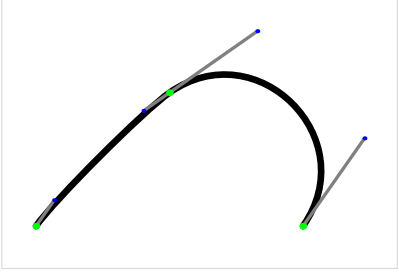
An example of that potential effect on the direction can be seen in the following code, where the two tension values for each segment are not identical:

```
path p;
p = (0,0)..tension 2 and 1 ..
    (100,100)..tension 1 and 2 ..
    (200,0);
detaileddraw p;
```



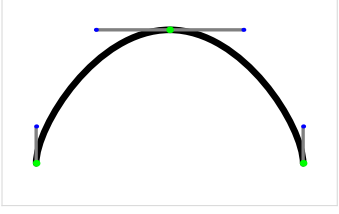
Another example is when all the segments do not have the same/complementary tension settings:

```
path p;
p = (0,0)..tension 2 ..
    (100,100)..
    (200,0);
detaileddraw p;
```

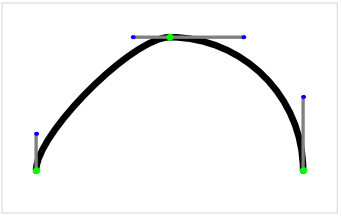


If we 'fixate' those examples by filling in explicit direction vectors:

```
path p;
p = (0,0){up}..tension 2 and 1 ..
    (100,100){right}..tension 1 and 2 ..
    {down}(200,0);
detaileddraw p;
```



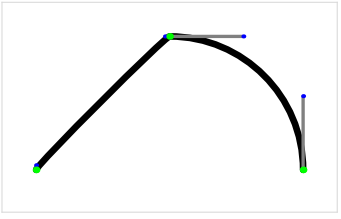
```
path p;
p = (0,0){up}..tension 2 ..
    (100,100){right}..
    {down}(200,0);
detaileddraw p;
```



the main effect of `tension` becomes clearer: it alters the control vectors. In both examples the length of the factors has been shortened where the `tension` was more than one.

In these cases, where all the directions are fixed already, every time the tension is multiplied by two, the length of the vector is divided by two. So with:

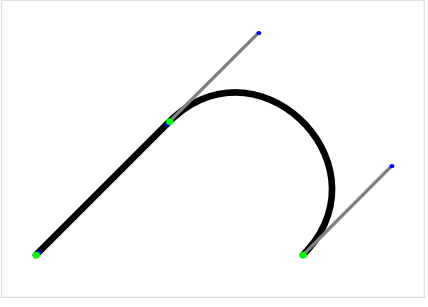
```
path p;
p = (0,0){up}..tension 16 ..
    (100,100){right}..
    {down}(200,0);
detaileddraw p;
```



the vector is now 1/16 of its 'normal' length, and it becomes almost a straight line.

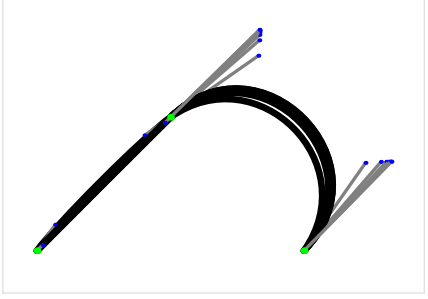
If there are multiple non-fixated directions, the vector changes become more complicated because in that case, the extra effect of the tension settings on the first segment is that the control vectors for the other segments become a bit longer:

```
path p;
p = (0,0)..tension 16 ..
    (100,100)..
    (200,0);
detaileddraw p;
```



This effect is more obvious when multiple values of tension are combined into a single image:

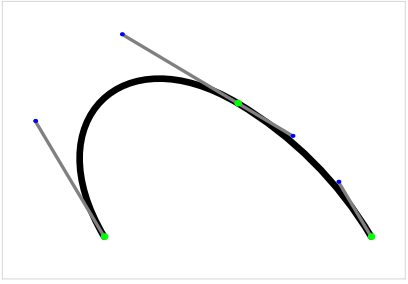
```
path p,q;
for i = 2 upto 8:
  p := (0,0)..tension i ..
    (100,100)..
    (200,0);
  detaileddraw p;
endfor
```



The high `tension` setting on the left segment has lowered the effective tension of the second segment.

The inverse effect of the vector shortening is also true, by the way. Tension values lower than one have the opposite effect:

```
path p;
p = (0,0)..tension 0.75 ..
    (100,100)..
    (200,0);
detaileddraw p;
```

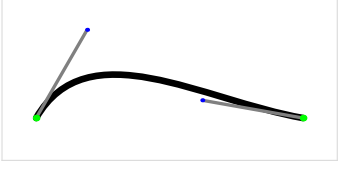


But lowering the tensions (and thus increasing the length of the vectors) also increases the chances that the internal calculations that control the behaviour of the curve become unpredictable. For that reason, the lowest value you are allowed to set `tension` to is `0.75`.

We have now seen various examples where the tension of the path can alter the directions of the path where it has not been set explicitly. But, as I wrote earlier, the directions of the path can also alter the values of the tensions in segments that do not have explicit values assigned to them.

Sometimes the latter results in sub-optimal curves, like in the following example where we may not want an inflection to happen:

```
path p;
p := (0,0){dir 60} ..
     {dir -10}(200,0);
detaileddraw p;
```



This last problem can be helped by using `tension atleast` which is a special case that sets a bottom limit for the final effective tension:

```
path p;
p := (0,0){dir 60}..tension atleast 1..
     {dir -10}(200,0);
detaileddraw p;
```

The combined calculations for `curl` and `tension` are at the core of how MetaPost manages to produce 'pleasing' curves with very little explicit set up by the user. But the fact that both calculations can actually effect each other means that sometimes the only way to be sure the result is exactly as desired is to verify it manually.

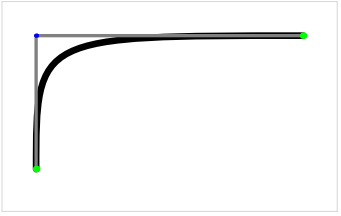
To wrap up the discussion of `tension`: here are two macros that are usually predefined:

```
def --- = .. tension infinity .. enddef;
def ... = .. tension atleast 1 .. enddef;
```

Explicit controls There may be cases where the internal calculations of MetaPost are not able or willing to create your desired output. And there may be other cases where you have a Bézier path converted from another input source that uses explicit control points.

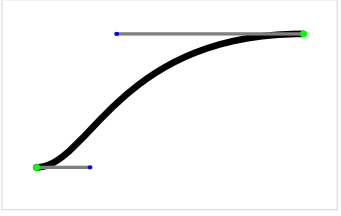
For such cases, MetaPost allows you to input explicit control point values, either by using a single point, like:

```
path p;
p = (0,0)..controls (0,100)..
    (200,100);
detaileddraw p;
```



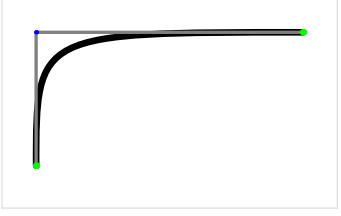
or by using two separate points:

```
path p;
p = (0,0)..controls (40,0) and (60,100)..
    (200,100);
detaileddraw p;
```



As with tensions, a single value is essentially the same as repeating that value. The first example is equivalent to:

```
path p;
p = (0,0)..controls (0,100) and (0,100)..
    (200,100);
detaileddraw p;
```

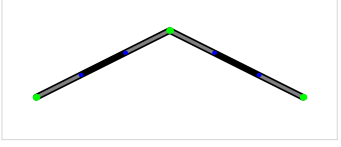


Handy to know:

- ☐ You cannot use `tension` and `controls` together in a single connector.
- ☐ Once processed, all path segments always use control points. Using explicit control points simply makes MetaPost skip all its own calculations.
- ☐ It follows that using explicit control points will overwrite any other `direction` or `curl` specification for the segment.

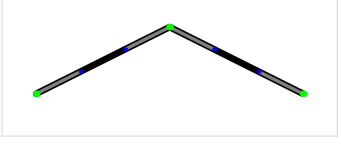
Path concatenation Just like with strings, it is possible to concatenate two paths by using the `&` operator:

```
path p;
p = (0,0)..(100,50) & (100,50)..(200,0);
detaileddraw p;
```



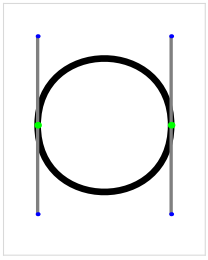
This only works if the left and right points are identical, and it is equivalent to

```
path p;
p = (0,0)..{curl 1}(100,50)..(200,0);
detaileddraw p;
```



Cyclic paths Creating a cyclic path is done by appending the `cycle` operator to the last connector:

```
path p;
p = (0,0)..(100,eps) .. cycle;
detaileddraw p;
```

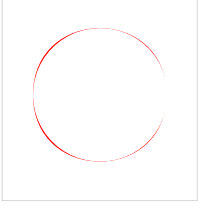


The `cycle` operator adds a reference back to the first point of the path being created, but it also adds a special marker to the internal path structure. Without it, the path is not considered to be cyclic, and you cannot use it with e.g. `fill`.

As can see, the example above produces a circular-looking path. This is the result of the automatic direction and tension calculations. The path is supposed to travel 360 degrees in total, and the internal calculations try to spread that curvature over the complete path, instead of producing two 180 degree turns with straight lines in between them. The nicest solution mathematically is to create two Bézier segments that have an amplitude that is half the distance between the two points, which is why you end up with a shape very similar to a circle.

It is not really a circle since Bézier curves simply cannot produce a perfect circle, but it is fairly close. The pen named `pencircle` actually is a perfect circle, so we can visually show the difference:

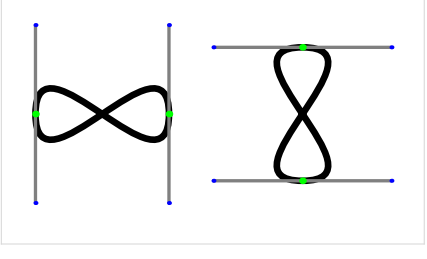
```
path p;
p = (0,0)..{down}(100,0) .. cycle;
fill p withcolor red;
draw origin shifted (50,0)
  withpen pencircle scaled 100
  withcolor white;
```



In all four quadrants there is a little bit of extra red that sticks out from behind the perfect invisible circle.

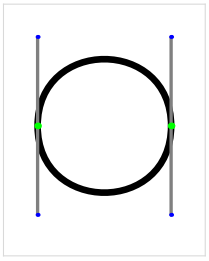
You may be wondering about the use of `eps` (defined as `.00049`) one example back. The reason is that while a two-point path can (and usually does) define a path that turns 360 degrees (by moving upward through the first point and then downward through the second point), it can also define a path that turns 0 degrees, where it goes up in both points. And it so happens that MetaPost decides on that second case if (and only if) the line through the two points is perfectly horizontal or vertical:

```
path p,q;
p = (0,50)..(100,50) .. cycle;
q = (200,0)..(200,100) .. cycle;
detaileddraw p;
detaileddraw q;
```



Another way to force the 360 degree case would be:

```
path p;
p = (0,0)..{down}(100,0) .. cycle;
detaileddraw p;
```



which is what I did in the overlay picture with the `pencircle`.

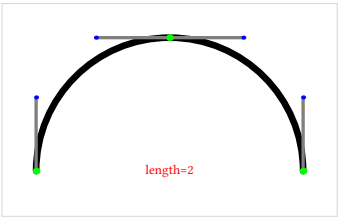
So now you have all the tools to define a path.

Path creation wrapup

To end this section, here is some 'handy to know' information that did not quite fit in the earlier parts of this section:

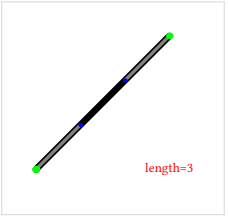
- the `length` of a path is the number of segments it consists of, which is equal to the number of explicit points for cyclic paths, and for non-cyclic paths, the same number minus one.

```
path p;
p = (0,0)..(100,100)..(200,0);
detaileddraw p;
labelat((100,0),
  "length=" & decimal length p) ;
```



I brought this up now instead of in the section on operations because it is important to know that 'empty' curve segments *do* count when you define a path, so:

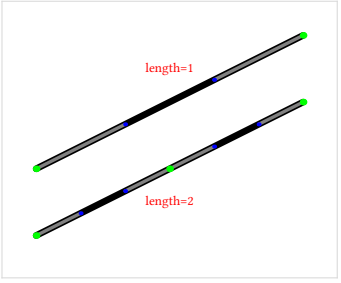
```
path p;
p = (0,0)..(0,0)..(0,0)..(100,100);
detaileddraw p;
labelat((100,0),
  "length=" & decimal length p) ;
```



defines a path of length 3.

- the `subpath` operator adds points at the beginning and end of the subpath if needed, so if you combine them back again you can get extra points:

```
path p, q;
q = (0,0)..(200,100);
p = subpath (0   , 0.5) of q &
    subpath (0.5, 1   ) of q ;
detaileddraw q shifted(0,25);
labelat((100,100),
    "length=" & decimal length q);
detaileddraw p shifted(0,-25);
labelat((100,0),
    "length=" & decimal length p);
```



In these examples I used labelat to put a bit of red text in the image. That is a macro I wrote in the preamble of this article. It will appear in a number of the following examples as well (along with draw_origin). I will not show you the definitions of those, they are just for illustrative purposes.

Defining a pair

In MetaPost, pairs are the building block of paths as well as the commonly used descriptions of other data that requires two values, like vectors and intersection times.

For reference, let's start with the formal definition:

⟨pair primary⟩ → ⟨pair variable⟩ | ⟨pair argument⟩
| (⟨numeric expression⟩ , ⟨numeric expression⟩)
| (⟨pair expression⟩)
| begingroup ⟨statement list⟩⟨pair expression⟩ endgroup
| ⟨numeric atom⟩ [⟨pair expression⟩ , ⟨pair expression⟩]
| ⟨scalar multiplication operator⟩⟨pair primary⟩
| point ⟨numeric expression⟩ of ⟨path primary⟩
| precontrol ⟨numeric expression⟩ of ⟨path primary⟩
| postcontrol ⟨numeric expression⟩ of ⟨path primary⟩
| penoffset ⟨pair expression⟩ of ⟨pen primary⟩
| penoffset ⟨pair expression⟩ of ⟨future pen primary⟩

⟨pair secondary⟩ → ⟨pair primary⟩
| ⟨pair secondary⟩⟨times or over⟩⟨numeric primary⟩
| ⟨numeric secondary⟩ * ⟨pair primary⟩
| ⟨pair secondary⟩⟨transformer⟩

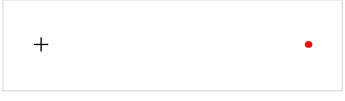
⟨pair tertiary⟩ → ⟨pair secondary⟩
| ⟨pair tertiary⟩⟨plus or minus⟩⟨pair secondary⟩
| ⟨path tertiary⟩ intersectiontimes ⟨path secondary⟩

⟨pair expression⟩ → ⟨pair tertiary⟩

Now, let's look at a few simple examples of defining pairs.

The most simple case is just a pair of numeric values:

```
pair a;
a = (200,0);
draw_origin;
drawdot a withcolor red;
```



The examples below will always print the 'active' pair in red, and the 'origin' point at (0,0) in black; and occasionally an extra dot or path is drawn as well. The red dot is always the target of the example.

The dot drawing is done by the drawdot macro, which is usually predefined in the macro package. In fact, a pair variable origin is usually also defined, so we will use that from now on as it is a little more readable.

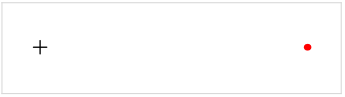
You can also define a pair from another pair variable:

```
pair a,b;
b = (200,0);
a = b;
draw_origin;
drawdot a withcolor red;
```

In fact, because of the way collections of equations are solved in MetaPost, you can invert the equation that gives a value to b and the equation that equates a to b. As

long as the equation is resolved before you try to draw the dot, the order of the equations is irrelevant. This is generally true in MetaPost but it bears repeating here because you will most often use this equation solving capability in the context of trying to resolve pairs:

```
pair a,b;
a = b;
b = (200,0);
draw_origin;
drawdot a withcolor red;
```



You can define a pair from an expression by adding parentheses:

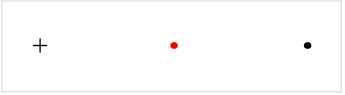
```
pair a;
a = ((100,0) + (100,0));
draw_origin;
drawdot (100,0);
drawdot a withcolor red;
```

The internal expression can then do all of the things from the full syntax. Besides addition, for example, you could also do multiplication:

```
pair a;
a = ((100,10) * 2);
draw_origin;
drawdot (100,10);
drawdot a withcolor red;
```

Sometimes, using an expression in this way is not the most elegant way of thinking about where the new point should be. That is why there is also an 'off-the-way' operation:

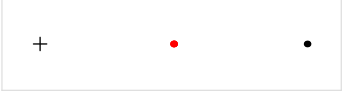
```
pair a;
a = .5[(0,0),(200,0)];
draw_origin;
drawdot (200,0);
drawdot a withcolor red;
```



The syntax rule says that the operator is an ⟨numeric atom⟩, which means that it can be either a numeric variable, or a numeric token (as in the example; this is typically a decimal fraction between zero and one), or it can be a numeric token followed by a `/` another numeric token (indicating an explicit fraction).

Here some equivalents of the previous example:

```
pair a,b,c;
i := .5;
a = i[(0,0),(200,0)];
b = 1/2[(0,0),(200,0)];
c = 5/10[(0,0),(200,0)];
draw_origin;
drawdot (200,0);
drawdot a withcolor red;
drawdot b withcolor red;
drawdot c withcolor red;
```



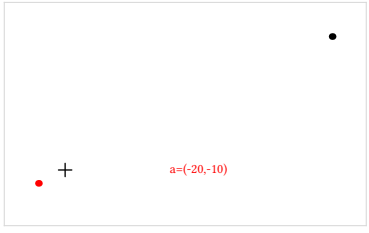
It is important to realize that while we informally call this the 'off-the-way' operator, it is really just a multiplier along a line where the first listed pair is at 'distance' zero

and the second listed pair is at 'distance' one. If the points are named a and b and the multiplier is x, then it calculates $a + x * (b - a)$. This means that the value can be outside of the zero-to-one range:

```
pair a;
a = 1.1[(0,0),(200,100)];
draw_origin;
drawdot (200,100);
drawdot a withcolor red;
labelat((100,0), "a=(220,110)")
```

It may equally well be negative:

```
pair a;
a = -0.1[(0,0),(200,100)];
draw_origin;
drawdot (200,100);
drawdot a withcolor red;
labelat((100,0), "a=(-20,-10)")
```



Note that there is no * allowed (or needed, depending on how you think about these things) between the value and the following square open bracket.

We can also define a new pair from a multiplication applied to another pair:

```
pair a;
a = .5(200,0);
draw_origin;
drawdot (200,0);
drawdot a withcolor red;
```

This example may look a bit odd, but it makes much more sense if the explicit pair is replaced by a predefined variable:

```
pair b; b = (200,0);
pair a;
a = .5b;
draw_origin;
drawdot b;
drawdot a withcolor red;
```

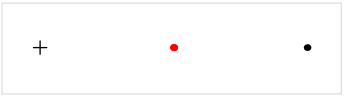
The formal syntax rules here are a little contrived, but the end result is that this operation is quite like the 'off-the-way' operator, except that variables are not allowed.

⟨scalar multiplication operator⟩ → ⟨plus or minus⟩
| ⟨numeric token primary not followed by + or - or a numeric token⟩

⟨numeric token primary⟩ → ⟨numeric token⟩/⟨numeric token⟩
| ⟨numeric token not followed by '/ numeric token' ⟩

Allowing a bare variable in the syntax here would confuse the language parser. It is simple enough to add a * to the input, but that does have a slightly different meaning to the MetaPost parser.

```
pair a,b,c;
i := .5;
% a = i(200,0); % not allowed
a = i*(200,0);
b = 1/2(200,0);
c = 5/10(200,0);
draw_origin;
drawdot (200,0);
drawdot a withcolor red;
drawdot b withcolor red;
drawdot c withcolor red;
```



The examples above with the `num/denom` operation need an extra bit of explanation, because the explicit `num/denom` form of a numeric token really is something different from just multiplying an expression. The explicit fraction here is never converted to a single fraction internally. The two specified values are both kept, which means that those calculations are a bit more precise.

The effect is not quite as obvious as the default `scaled` number system of plain MetaPost when using the new `double` number system, but the difference is still important sometimes.

If you run the following in plain MetaPost:

```
pair a,b;
a = 1/5(100,100);
b = 1/5*(100,100);
show a;
show b;
```

it will report the following:

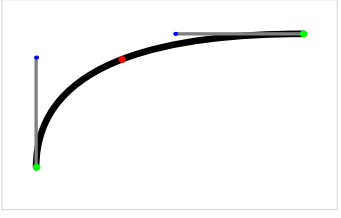
```
>> (20,20)
>> (19.9997,19.9997)
```

Because the first equation resolves to $(1 * 100/5, 1 * 100/5)$, which gives the perfect result of `(20,20)`, while the second equation is $((13107/65536) * 100, (13107/65536) * 100)$. The `(13107/65536)` part is the internal representation of `1/5` as a decimal fraction.

This difference between the exact value and approximation of a fraction happens in the `decimal` number system as well, although it is far less obvious there thanks to the higher precision.

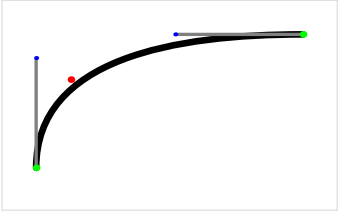
The next two examples deal with defining a new pair based on some part of a path. First, you take any specific point along a path (it does not have to be an integer value):

```
path p;
pair a;
p = (0,0){up}..{right}(200,100);
a = point 0.5 of p;
detaileddraw p;
drawdot a withcolor red;
```



Or you can use one of the control points of a point along that path:

```
path p;
pair a;
p = (0,0){up}..{right}(200,100);
a = precontrol 0.5 of p;
detaileddraw p;
drawdot a withcolor red;
```



There is also `postcontrol`, of course. This comes with a warning: the actual path will change because it first creates a knot at the specified place along the path, as is shown in this example.

What you are actually getting from `precontrol` is this:

```
path p,q;
pair a;
p = (0,0){up}..{right}(200,100);
q = subpath (0,0.5) of p
    & subpath(0.5,1) of p;
a = precontrol 1 of q;
detaileddraw q;
drawdot a withcolor red;
```

This effect does not happen if you use an integer `point` along the path, because in this case, MetaPost does not have to bisect the path.

From a path intersection (indirectly):

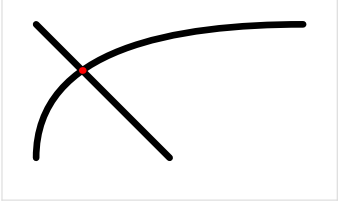
```
path p,q;
pair a;
p = (0,0){up}..{right}(200,100);
q = (0,100)..(50,50)..(100,0);
a = p intersectiontimes q;
draw p withcolor .5;
draw q withcolor .5;
drawdot a withcolor red;
```

No, that is not a typographical error in the example!

The `intersectiontimes` returns two time values along the paths, encoded as a `pair`. In this case, that is `(0.35608,0.69121)`. There are two separate points referenced in this single pair – `point 0.35608` along `p`, and `point 0.69121` along `q` – but neither value represents a point individually.

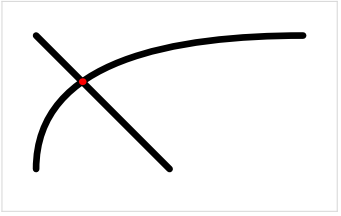
If you want to have an actual point, you have to fetch it from the path using the `point` operator:

```
path p, q;
pair a, b;
p = (0,0){up}..{right}(200,100);
q = (0,100)..(50,50)..(100,0);
a = p intersectiontimes q;
b = point (xpart a) of p;
draw p; draw q;
drawdot b withcolor red;
```



Normally there is a predefined macro `intersectionpoint` that you can use as a drop-in replacement for `intersectiontimes`, like this:

```
path p, q;
pair a;
p = (0,0){up}..{right}(200,100);
q = (0,100)..(50,50)..(100,0);
a = p intersectionpoint q;
draw p; draw q;
drawdot a withcolor red;
```



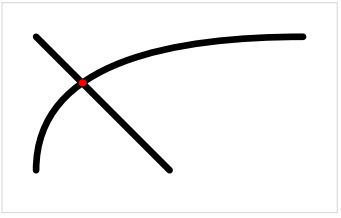
But you need to be careful using it because, owing to the definition of `intersectionpoint`, you will usually get a point that is not on *either* path. The typical definition of this macro tries to give you a point that is very close to both of the two points. The simplified definition looks like this:

```
secondarydef p intersectionpoint q =
  begingroup
    save x_,y_;
    (x_,y_)=p intersectiontimes q;
    .5[point x_ of p, point y_ of q]
  endgroup
enddef;
```

It does this 'off-the-way' operation because there is no guarantee that the separate times along the paths will result in a single coincident point; even within the precision limits of the `scaled` number system. There are various limitations in the algorithm in MetaPost that is used to find the time values (e.g. it stops bisection of the path long before the maximum precision is reached) that are intended to save on memory usage and processing time. As a result, you rarely get perfect time values returned.

In our example above, the two points are visually indistinguishable at the normal zoom level:

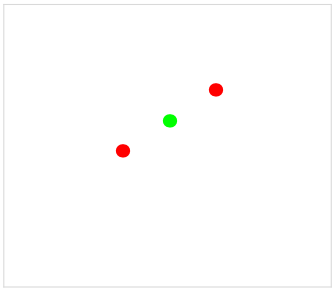
```
path p, q;
pair a, b, c;
p = (0,0){up}..{right}(200,100);
q = (0,100)..(50,50)..(100,0);
a = p intersectiontimes q;
b = point (xpart a) of p;
c = point (ypart a) of q;
draw p; draw q;
drawdot b withcolor red;
drawdot c withcolor red;
```



But on close inspection, `b` is `(34.56109,65.44007)`, whereas `c` is `(34.56039,65.439605)`. The returned expression by `intersectionpoint` is therefore `(34.56074,65.43984)`.

Zoomed in, it looks like this:

```
...
b = point (xpart a) of p;
c = point (ypart a) of q;
draw p; draw q;
drawdot b withcolor red;
drawdot c withcolor red;
drawdot (p intersectionpoint q)
          withcolor green;
```



If there are no intersections, `intersectiontimes` returns `(-1,-1)`.

If there are multiple intersections, it normally returns the first one along the left-side path.

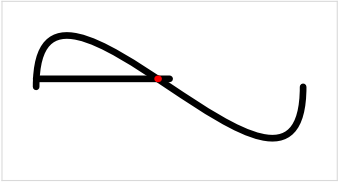
However, it is actually possible for there to be multiple intersections within a single curve segment (in other words: in the curve section 'between' two of knot points of one of the paths). In this case, MetaPost will return the 'smallest' combination of times along both paths. Here is an example of where that can happen:

```
pair a;
path p, q;
p := (0,0){up}..{up}(200,0);
q := (200,6)..(0,6);
a := p intersectiontimes q;
draw p; draw q;
drawdot (point (xpart a) of p)
          withcolor red;
```

In this case, it returns the first intersection along `p`, as expected. Note that `q` is moving to the left, not the right, so it is actually the second intersection along `q`.

But if we make `q` shorter:

```
pair a;
path p, q;
p := (0,0){up}..{up}(200,0);
q := (100,6)..(0,6);
a := p intersectiontimes q;
draw p; draw q;
drawdot (point (xpart a) of p)
          withcolor red;
```



it will jump to the *second* intersection on `p` (and thus the first on `q`) instead.

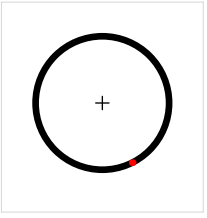
The intersection times along `p` are the same in both cases: `0.01573` and `0.46989`.

But the intersections along `q` are different. In the first example they happen roughly at the times `0.55` and `0.98` (remember, it is coming from the right), and in the second example they are more like `0.09` and `0.96`.

In the second example, MetaPost returns the result `(0.46989,0.09)` because if you add these two times up, the result is less than the addition of `(0.01573,0.96)`. In the first example, the total of `(0.46989,0.55)` was a little bit more than `(0.01573,0.98)` instead of less, so it returned the other option instead.

The last way to define a pair is using a 'pen offset':

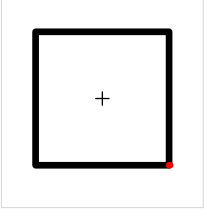
```
path p;
pair a;
a = penoffset (1,0.5) of
      pencircle scaled 100;
p = makepath pencircle scaled 100;
draw p;
draw_origin;
drawdot a withcolor red;
```



The primitive operator `penoffset` returns the 'offset' along the pen in which the pen travels in the direction vector given by its argument (in this case, that is a direction of approximately 26.5 degrees, because the pair is treated as an angular vector). The return value is the x, y offset of the edge of the pen to its center at the moment it is moving in that direction. In the example above the center of the pen is the `origin` so the offset is simply a point in the coordinate system. That point is roughly `(22.36,-44.72)`, where the pen outline moves up at an angle of approximately 26.5 degrees (it helps to know that `pen`s rotate counterclockwise).

For polygonal pens, the results can be a bit confusing because the corner points are treated as if they have all directions between the incoming and outgoing angles.

```
path p;
pair a;
a = penoffset (1,0.5) of
      pensquare scaled 100;
p = makepath pensquare scaled 100;
draw p;
draw_origin;
drawdot a withcolor red;
```



Pens

After all this stuff about `path`s and `pair`s, `pen`s are surprisingly simple. There is just not that much you can do with pens. And in MetaPost, pens are even simpler than in Metafont, because the code that converts elliptical pens into bitmaps was not needed in MetaPost, which makes the syntax cleaner.

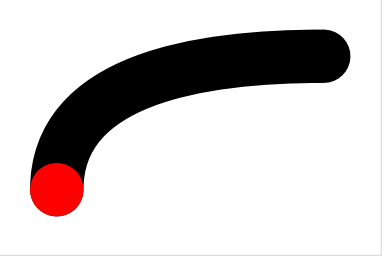
Pens are the objects that are used to 'trace' paths when you use `draw` or `filldraw` (or rather the underlying primitive `addto`). MetaPost has two different kind of pens: pens that derived from a circle (a.k.a. elliptical pens) and pens that are based on a convex cycle of straight segments (a.k.a. polygonal pens).

The adjusted syntax rule is:

⟨pen primary⟩ → ⟨pen variable⟩ | ⟨pen argument⟩
| (⟨pen expression⟩)
| begingroup ⟨statement list⟩⟨pen expression⟩ endgroup
| nullpen
| pencircle
| makepen ⟨path primary⟩
⟨pen secondary⟩ → ⟨pen primary⟩
| ⟨pen secondary⟩⟨transformer⟩
⟨pen tertiary⟩ → ⟨pen secondary⟩
⟨pen expression⟩ → ⟨pen tertiary⟩

We will get to the ⟨transformer⟩ in a later section (as for paths and pairs). And you should be familiar by now with the ⟨variable⟩ , ⟨argument⟩, ⟨expression⟩, and `begingroup` ⟨...⟩ `endgroup` parts. So what is left is defining a pen based on the built-in `pencircle`:

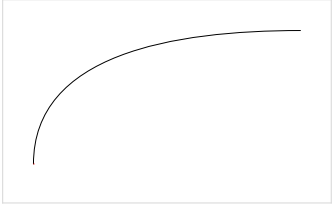
```
pen mypen;
mypen = pencircle scaled 40;
draw (0,0){up}..{right}(200,100)
  withpen mypen;
fill makepath mypen withcolor red;
```



You can do all sorts of elliptical pens this way by using the ⟨transformer⟩ to rotate and scale the `pencircle`. Internally, an elliptical pen is a perfect ellipse. It is never converted into separate path segments unless the user asks for it (by using `makepath` to make the pen itself drawable, as we do in these examples).

Defining a pen based on the built-in `nullpen`:

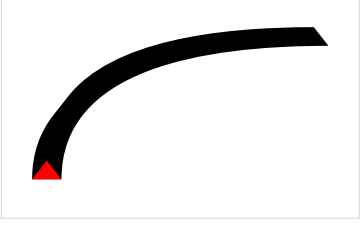
```
pen mypen;
mypen = nullpen scaled 4000;
draw (0,0){up}..{right}(200,100)
  withpen mypen;
fill makepath mypen withcolor red;
```



But this is only useful to clear an existing pen, as the `nullpen` is a pen with no dimensions.

Finally, you can define a `pen` from a path:

```
pen mypen;
path p;
p = (-11,0)--(0,14)--(11,0)--cycle;
mypen = makepen p;
draw (0,0){up}..{right}(200,100)
  withpen mypen;
fill makepath mypen withcolor red;
```



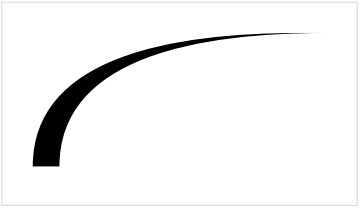
The result of `makepen` is always a polygonal pen. It is not possible to construct elliptical pens in this way as they *have* to be based on `pencircle`.

Some handy things to know:

- ☐ `makepen` always converts `..` to `--`.
- ☐ Pens are always convex; `makepen` will silently enforce this by ignoring concaveness-inducing points.
- ☐ While elliptical pens are created by transforming `pencircle`, it can sometimes be useful to create a polygonal pen with many vertices as an approximation, for example, for the `envelope` operation that we saw earlier.
- ☐ MetaPost's pens always travel in an counter-clockwise direction, even if the input path to `makepen` was clockwise.

MetaPost does not have a true linear pen, but it is easy to approximate one:

```
pen mypen;
path p;
p = (-10,0)--(10,0)--cycle;
mypen = makepen p;
draw (0,0){up}..{right}(200,100)
  withpen mypen;
fill makepath mypen withcolor red;
```



A final hint about defining pens for reuse: when you make a special pen shape inside of a macro file that will be reused, it is good practice to give it a clear name, and to place the pen around the origin with one of its major sizes close to 1. This makes it easier for other macros to build upon the defined pen by rotating and scaling it.

So, instead of the earlier example, which was equivalent to:

```
pen mypen;
mypen = makepen((-11,0)--(0,14)--(11,0)--cycle);
```

Use this:

```
pen penpyramid;
penpyramid = makepen((-0.5,0)--(0,14/22)--(0.5,0)--cycle);
```

Transformations

Transformations make MetaPost much more versatile. Here is the formal syntax definition of everything related to transformations:

⟨transform primary⟩ → ⟨transform variable⟩ | ⟨transform argument⟩
| (⟨transform expression⟩)
| begingroup ⟨statement list⟩⟨transform expression⟩ endgroup

⟨transform secondary⟩ → ⟨transform primary⟩
| ⟨transform secondary⟩⟨transformer⟩

⟨transform tertiary⟩ → ⟨transform secondary⟩

⟨transform expression⟩ → ⟨transform tertiary⟩

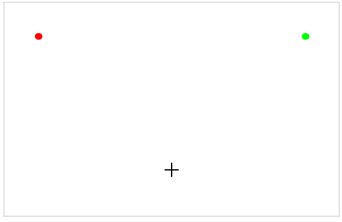
⟨transformer⟩ → rotated⟨numeric primary⟩
| scaled ⟨numeric primary⟩
| shifted ⟨pair primary⟩
| slanted ⟨numeric primary⟩
| transformed ⟨transform primary⟩
| xscaled ⟨numeric primary⟩
| yscaled ⟨numeric primary⟩
| zscaled ⟨pair primary⟩

Whenever you use an object of type `path`, `pair` or `pen` (as well as `picture` and `transform` itself) in a MetaPost expression at the secondary level, you are allowed to transform it using a ⟨transformer⟩.

The following transformation options apply to all those object types, but I will only show `pair`s as examples to keep it simple. In these examples, the green dot is the original, the red dot is the transformed one, and the black cross is at the origin (`0,0`).

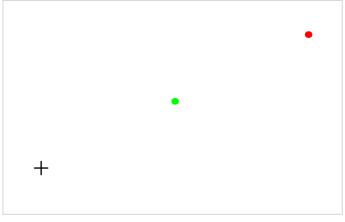
`rotated` works counter-clockwise around the origin:

```
pair a;
a = (100,100) rotated 90;
draw_origin;
drawdot (100,100) withcolor green;
drawdot a withcolor red;
```



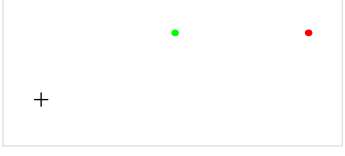
`scaled` multiplies the separate components:

```
pair a;
a = (100,50) scaled 2;
draw_origin;
drawdot (100,50) withcolor green;
drawdot a withcolor red;
```



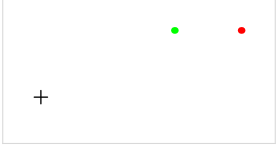
`shifted` moves things around:

```
pair a;
a = (100,50) shifted (100,0);
draw_origin;
drawdot (100,50) withcolor green;
drawdot a withcolor red;
```



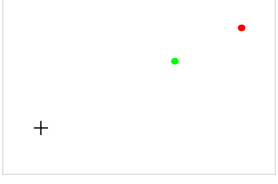
`slanted` slants things by adding some multiple of the y value to the x value:

```
pair a;
a = (100,50) slanted 1;
draw_origin;
drawdot (100,50) withcolor green;
drawdot a withcolor red;
```



`transformed` applies a complete 6-variable transformation matrix:

```
pair a;
transform t;
t := identity scaled 1.5;
a = (100,50) transformed t;
draw_origin;
drawdot (100,50) withcolor green;
drawdot a withcolor red;
```



Because `transformed` is at the core of the transformation commands, this is a good moment to delve a little deeper into what transformations are and do in MetaPost. A `transform` variable consists of six parts: `xpart`, `ypart`, `xxpart`, `xypart`, `yxpart` and `yypart`. This is similar to a `pair` variable, only there are more parts.

Transformations are just a shorthand notation for applying a set of operations on an object. For pairs, the expression `(x,y) transformed t` converts the pair (x,y) into the pair $(t_x + xt_{xx} + yt_{xy},\ t_y + xt_{yx} + yt_{yy})$.

Interestingly, there is no direct way to define a variable of type `transform`.

Even the transform `identity` is not actually a primitive, but it is defined in a somewhat curious way in the plain MetaPost macros:

```
transform identity;
for z=origin,right,up:
  z transformed identity = z;
endfor
```

The three equations in the `for` loop resolve all six parts of the transform object together:

```
origin transformed identity = origin;
right transformed identity = right;
up transformed identity = up;
```

Remember that `origin`, `right`, and `up` are defined pairs. Those are already known at this point in the macro loading process, so the three formulas are actually:

```
(0,0) transformed identity = (0,0);
(1,0) transformed identity = (1,0);
(0,1) transformed identity = (0,1);
```

And because they are equations, they can be inverted to set up the six parts of the transformation:

```
(0,0) transformed identity = (0,0);
```

expands to:

$$(t_x + x * t_{xx} + y * t_{xy}, t_y + x * t_{yx} + y * t_{yy}) = (0, 0)$$

with x and y already known to be 0, it is easy to see this reduces to:

$$(t_x, t_y) = (0, 0)$$

In the next equation, these two values are now also known, so:

```
(1,0) transformed identity = (1,0);
```

is really

$$(0 + 1 * t_{xx} + 0 * t_{xy}, 0 + 1 * t_{yx} + 0 * t_{yy}) = (1, 0)$$

or, simplified:

$$(t_{xx}, t_{yx}) = (1, 0)$$

So the t_{xx} part must be 1 and the t_{yx} part 0. At the last step, there are only two variables left to calculate. The final equation:

```
(0,1) transformed identity = (0,1);
```

wraps this up with:

$$(0 + 0 * 1 + 1 * t_{xy}, 0 + 0 * 0 + 1 * t_{yy}) = (0, 1)$$

$$(t_{xy}, t_{yy}) = (0, 1)$$

So t_{xy} must be 0 and t_{yy} must be 1.

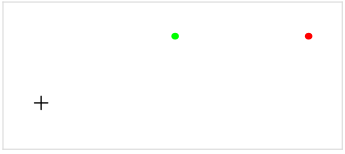
The `identity` transformation could also have been defined like this:

```
transform identity;
xpart identity  = ypart identity  = 0;
xxpart identity = yypart identity = 1;
xypart identity = yxpart identity = 0;
```

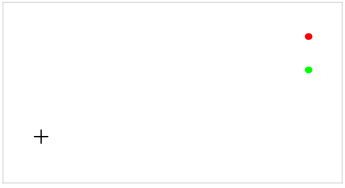
but those six equations are not nearly as cute or fun to explain.

We already saw the most important shorthand ⟨transformer⟩s, but there are three more:

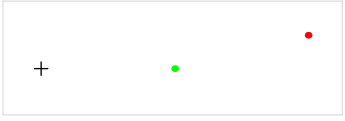
```
pair a;
a = (100,50) xscaled 2;
draw_origin;
drawdot (100,50) withcolor green;
drawdot a withcolor red;
```



```
pair a;
a = (200,50) yscaled 1.5;
draw_origin;
drawdot (200,50) withcolor green;
drawdot a withcolor red;
```



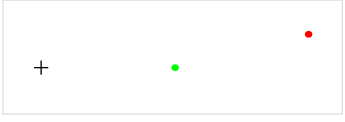
```
pair a;
a = (100,0) zscaled (2,0.25);
draw_origin;
drawdot (100,0) withcolor green;
drawdot a withcolor red;
```



The `zscaled` operation may seem a bit weird.

One way of looking at it is that it treats its argument as a vector. It then rotates over the angle of that vector and scales by the length of it:

```
pair a;
a = (100,0)
    rotated angle (2,0.25)
    scaled (2++0.25);
draw_origin;
drawdot (100,0) withcolor green;
drawdot a withcolor red;
```



Another way of looking at `zscaled` is that it performs complex number multiplication. If the argument is u, v it converts x, y into $xu - yv, xv + yu$.

To wrap up our discussion of transformations, so things that are handy to remember:

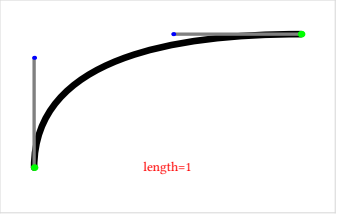
- ☐ You can chain transformers, they are processed left to right.
- ☐ There is no direct assignment syntax for `transform` type definitions: you have to modify an existing transform, build one using explicit ⟨transformer⟩ equations, or assign each of the six parts using separate equations.
- ☐ Don't forget to add groupings if you are mixing `pair` and `path` in the same expression.

Path operations

Now let's look at the operations you can do on paths.

Find the length of a path:

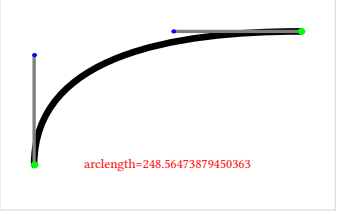
```
path p; numeric d;
p = (0,0){up}..{right}(200,100);
d = length p;
detaileddraw p;
labelat((100,0), "length="&decimal d);
```



The length operator returns the number of segments. That is one less than the number of defining points, unless the path is a cycle.

Find the drawn length of a path:

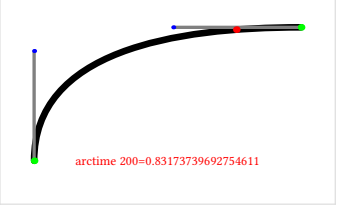
```
path p; numeric d;
p = (0,0){up}..{right}(200,100);
d = arclength p;
detaileddraw p;
labelat((100,0), "arclength="&decimal d);
```



This returns the total length of the actual curve(s).

Find a specific drawn time of a path:

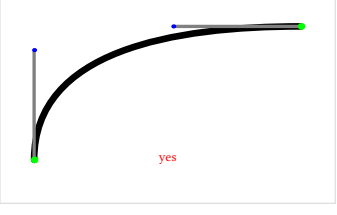
```
path p; numeric d;
p = (0,0){up}..{right}(200,100);
d = arctime 200 of p;
detaileddraw p;
labelat((100,0),
        "arctime 200="&decimal d);
drawdot (point d of p) withcolor red;
```



This returns the time along the path at which the arclength is the specified value.

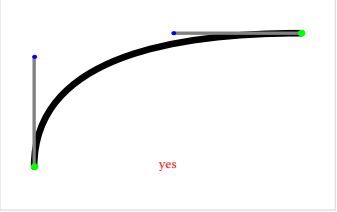
Test if a variable is a path:

```
path p;
p = (0,0){up}..{right}(200,100);
detaileddraw p;
if path p:
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```



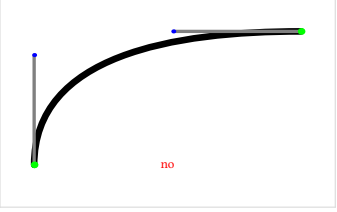
The if command tests the type of the following expression. This means that single pairs fail even though they are valid as path declarations (due to the automatic conversion into a path when an assignment takes place). But on the other hand, it means that you can use an explicit expression:

```
path p;
p = (0,0){up}..{right}(200,100);
detaileddraw p;
if path ((0,0){up}..{right}(200,100)):
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```



Test if a variable is a cyclic path:

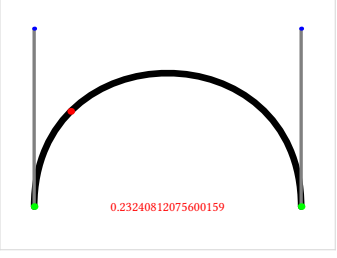
```
path p;
p = (0,0){up}..{right}(200,100);
detaileddraw p;
if cycle p:
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```



Only paths created with `cycle` are considered cyclic. Paths that just so happen to end at the same coordinates as they started are not considered a cycle by MetaPost. There is an implied `if path`, so you do not have to test for that separately.

Find the time at which a path moves in a certain direction:

```
path p; numeric d;
p = (0,0){up}..{down}(200,0);
detaileddraw p;
d = directiontime (1,1) of p;
labelat((100,0), decimal d);
drawdot (point d of p) withcolor red;
```



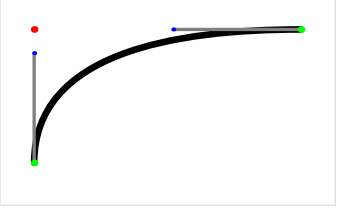
Side note: in this example it is obvious that a Bézier curve is *not* the same as a circular arc. If they were, the return value would have been exactly `0.25`.

Some other things of note about `directiontime`:

- ☐ the `pair` argument is treated as a direction vector
- ☐ if the path never travels in that direction, the return value is `-1`
- ☐ if the path travels multiple times in that direction, the first of those is returned.
- ☐ corner points are assumed to have all directions between the incoming and outgoing angles simultaneously.

Finally, it is possible to find any one of the bounding box points of a path:

```
path p; pair a;
p = (0,0){up}..{right}(200,100);
detaileddraw p;
a = ulcorner p;
drawdot a withcolor red;
```



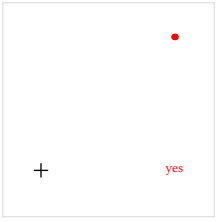
Also defined are the complementing primitives `llcorner`, `lrcorner`, and `urcorner`.

Pair operations

Now let's look at the operations you can do on `pair`s.

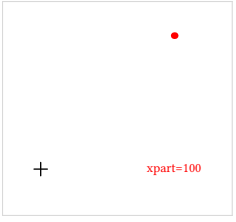
Test if a variable is a pair:

```
pair a;
a = (100,100);
draw_origin;
drawdot a withcolor red;
if pair a:
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```



Get the x or y part:

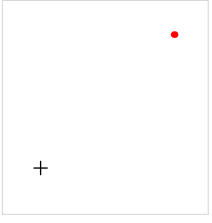
```
pair a; numeric d;
a = (100,100);
d = xpart a;
draw_origin;
drawdot a withcolor red;
labelat((100,0), "xpart=" & decimal d);
```



Of course there is also a matching `ypart` operation.

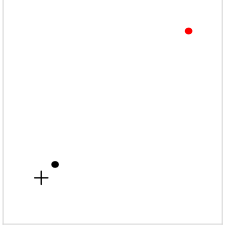
You can multiply or divide a pair by a numeric:

```
pair a;
a = (50,50) * 2;
draw_origin;
drawdot a withcolor red;
```



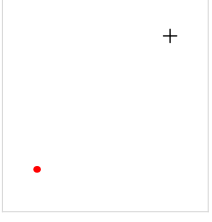
You can add or subtract another pair:

```
pair a,b;
b = (10,10);
a = (100,100) + b;
draw_origin;
drawdot b;
drawdot a withcolor red;
```



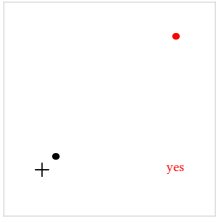
You can negate a pair:

```
pair a;
a = -(100,100);
draw_origin;
drawdot a withcolor red;
```



You can compare a pair to another pair:

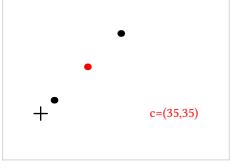
```
pair a,b;
b = (10,10);
a = (100,100);
draw_origin;
drawdot b;
drawdot a withcolor red;
if a > b:
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```



Pairs are first compared using the `xpart` values. If these are equal, the `ypart` values are compared.

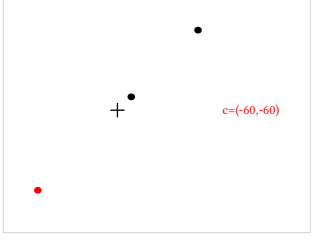
You can mediate between two pairs using the off-the-way operation:

```
pair a,b,c;
a = (10,10);
b = (60,60);
c = 0.5[a,b];
draw_origin;
drawdot a;
drawdot b;
drawdot c withcolor red;
labelat((100,0), "c=(35,35)")
```



When using mediation with negative values, you have to keep in mind that unary minus binds less forcefully than mediation:

```
pair a,b,c;
a = (10,10);
b = (60,60);
c = -1[a,b];
draw_origin;
drawdot a;
drawdot b;
drawdot c withcolor red;
labelat((100,0), "c=(-60,-60)")
```

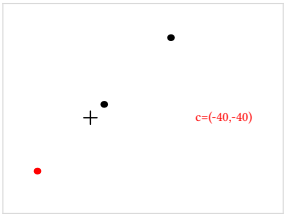


The result here is `(-60,-60)` because the mediation is processed first (using the positive value of `1`):

$$a + x * (b - a) \rightarrow 10 + 1 * (60 - 10) \rightarrow 10 + 60 - 10$$

And not until after this has been processed to `(60,60)` is the pair then negated, whereas in:

```
pair a,b,c;
a = (10,10);
b = (60,60);
c = (-1)[a,b];
draw_origin;
drawdot a;
drawdot b;
drawdot c withcolor red;
labelat((100,0), "c=(-40,-40)")
```

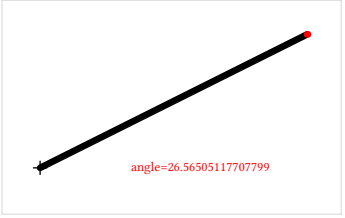


The result is `(-40,-40)` because the mediation is processed with the value `-1`:

$$a + x * (b - a) \rightarrow 10 + (-1) * (60 - 10) \rightarrow 10 - 60 + 10$$

For the last of the pair operations, when looking at a pair as a vector, it is often handy to know the angle:

```
pair a; numeric d;
a = (200,100);
d = angle a;
draw_origin;
draw (origin--a);
drawdot a withcolor red;
labelat((120,0), "angle="&decimal d)
```



Pen operations

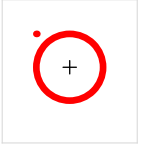
Now let's look at operations you can do on pens. There are just a few of those, because pens as independent objects are not very useful.

Test if a variable is a pen:

```
pen mypen;
mypen = pencircle scaled 50;
draw_origin;
draw makepath mypen withcolor red;
if pen mypen:
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```

And, just like for paths, it is possible to find any one of the bounding box points of a pen:

```
pen mypen; pair a;
mypen = pencircle scaled 50;
draw_origin;
a = ulcorner mypen;
draw makepath mypen withcolor red;
drawdot a withcolor red;
```



Also defined are the complementing primitives `llcorner`, `lrcorner`, and `urcorner`.

Transform operations

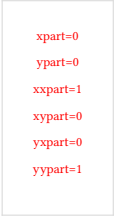
Now let's look at the operations you can do on `transform`s. Like with pens, there are not a whole lot of them.

Test if a variable is a transform:

```
if transform identity:
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```


Extract any of the constituent parts:

```
transform id;
id = identity;
labelat((0,100), "xpart=" & decimal xpart  id);
labelat((0,80),  "ypart=" & decimal ypart  id);
labelat((0,60), "xxpart=" & decimal xxpart id);
labelat((0,40), "xypart=" & decimal xypart id);
labelat((0,20), "yxpart=" & decimal yxpart id);
labelat((0,0),  "yypart=" & decimal yypart id);
```



Compare a transform with another transform:

```
transform T,V;
T = identity;
V = T scaled 2;
if T<V:
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```


Comparison of transforms tests `xpart`, `ypart`, `xxpart`, `xypart`, `yxpart`, `yypart` consecutively. Note that this assigns the most importance to the translation part of the transformation, which may not be how you think about transformation matrix sizing. In some cases it may be better to compare the `xxpart` and `yypart` explicitly.

Wrap-up

This article documents all of the primitive operations relating to paths, pairs, pens, and transforms.

Normally, one would use MetaPost with a preloaded macro package, and such a package will of course define extra operators, functions, predefined variables, et cetera.

For example, `plain.mp` defines all the extra identifiers already mentioned earlier in this article, but also the pair constants `left` and `down`, the path constants `quartercircle`, `halfcircle`, `fullcircle` and `unitsquare`, the pen constants `pensquare`, `penrazor` and `penspeck`, and unary operators `dir` and `unitvector` for pairs (vectors), `inverse` for transforms, and `center` for paths. And that is just in the first 200 lines or so of that macro package.

This article does not mention all of those additional definitions on purpose. It is very long already, and adding just the definitions from `plain.mp` would easily add another 20 pages, let alone the number of additions in MetaFun. For practical use of those macro packages, you will have to look at their documentation. The goal here is to show you the underpinnings beneath all of those smart macros. Nothing more, and nothing less.

Taco Hoekwater