MAPS

NUMMER 53 • VOORJAAR 2023

REDACTIE

Frans Goddijn, gangmaker
Taco Hoekwater

NTG NEDERLANDSTALIGE TEX GEBRUIKERSGROEP

NEDERLANDSTALIGE TEX GEBRUIKERSGROEP

De **Nederlandstalige TEX Gebruikersgroep (NTG)** is een vereniging die tot doel heeft de kennis en het gebruik van TEX te bevorderen. De NTG fungeert als een forum voor nieuwe ontwikkelingen met betrekking tot computergebaseerde documentopmaak in het algemeen en de ontwikkeling van 'TEX and friends' in het bijzonder. De doelstellingen probeert de NTG te realiseren door onder meer het uitwisselen van informatie, het organiseren van conferenties en symposia met betrekking tot TEX en daarmee verwante programmatuur.
De NTG biedt haar leden ondermeer:

- ☐ Tweemaal per jaar een NTG-bijeenkomst.
- ☐ Het NTG-tijdschrift MAPS.
- ☐ De 'TEX Live'-distributie op DVD/CDROM inclusief de complete CTAN software-archieven.
- ☐ Verschillende discussielijsten (mailing lists) over TEX-gerelateerde onderwerpen, zowel voor beginners als gevorderden, algemeen en specialistisch.
- ☐ De FTP server `ftp.ntg.nl` waarop vele honderden megabytes aan algemeen te gebruiken 'TEX-producten' staan.
- ☐ De WWW server `www.ntg.nl` waarop algemene informatie staat over de NTG, bijeenkomsten, publicaties en links naar andere TEX sites.
- ☐ Korting op (buitenlandse) TEX-conferenties en -cursussen en op het lidmaatschap van andere TEX-gebruikersgroepen.

Lid worden kan door overmaking van de verschuldigde contributie naar de NTG-giro (zie links); vermeld IBAN zowel als SWIFT/BIC en selecteer shared cost. Daarnaast dient via `www.ntg.nl` een informatieformulier te worden ingevuld. Zonodig kan ook een papieren formulier bij het secretariaat worden opgevraagd.
De contributie bedraagt € 35. Voor studenten geldt een tarief van € 18. Dit geeft alle lidmaatschapsvoordelen maar *geen stemrecht*. Een bewijs van inschrijving is vereist. Een gecombineerd NTG/TUG-lidmaatschap levert een korting van 10% op beide contributies op. De prijs in euro's wordt bepaald door de dollarkoers aan het begin van het jaar. De ongekorte TUG-contributie is momenteel $105.

Afmelding kan met ingang van het volgende kalenderjaar door opzegging per e-mail aan de penningmeester.

MAPS bijdragen kunt u opsturen naar `maps@ntg.nl`, bij voorkeur in LATEX- of ConTEXt formaat. Bijdragen op alle niveaus van expertise zijn welkom.

Productie. De Maps wordt gezet met behulp van een LATEX class file en een ConTEXt module. Het pdf bestand voor de drukker wordt aangemaakt met behulp van pdftex 1.40 en luametatex 2.09 draaiend onder MacOS X 13. De gebruikte fonts zijn Linux Libertine, het niet-proportionele font Inconsolata, schreefloze fonts uit de Latin Modern collectie, en de Euler wiskunde fonts, alle vrij beschikbaar.

TEX is een door professor Donald E. Knuth ontwikkelde 'opmaaktaal' voor het letterzetten van documenten, een documentopmaaksysteem. Met TEX is het mogelijk om kwalitatief hoogstaand drukwerk te vervaardigen. Het is eveneens zeer geschikt voor formules in mathematische teksten.
Er is een aantal op TEX gebaseerde producten, waarmee ook de logische structuur van een document beschreven kan worden, met behoud van de letterzet-mogelijkheden van TEX. Voorbeelden zijn LATEX van Leslie Lamport, $\mathcal{AMS}$-TEX van Michael Spivak, en ConTEXt van Hans Hagen.

Contents

Redactioneel

Dit is een speciale uitgave van de Maps met daarin vijf artikelen van de hand van Taco Hoekwater over diverse fundamentele zaken in MetaPost.

Door middel van de Maps willen we u op de hoogte houden van ontwikkelingen, ook om daarmee onze leden te danken voor hun trouwe steun aan de TEX ontwikkelaars. Verder bieden we ruimte aan lezers die anderen laten delen in hun ervaringen met TEX, MetaPost, fonts en aanverwanten. Aarzel dus niet ons artikelen te sturen. Een halve pagina is al heel leuk, meer mag ook, graag zelfs. Het hoeft geen ‚zware kost' te zijn want het is voor lezers bijvoorbeeld al heel interessant te lezen hoe anderen TEX gebruiken. Dus een artikeltje als „dit doe ik met TEX, zo doe ik dat en nu kun jij het ook" is zeer welkom!

Hoewel het internet tegenwoordig een belangrijke bron van informatie is, blijft papier een functie vervullen binnen de vereniging. Dat past immers bij TEX!

Veel leesplezier,

Uw redactie

Introduction

What you have here is a series of articles about details of the MetaPost programming language.

The target audience of these articles are users that are already somewhat familiar with simple graphics in MetaPost but want to have a clearer understanding of the language to make better use of its possibilities.

Each of the articles discusses a specific subsystem and together they should provide a solid base for improving the reader's knowledge of MetaPost.

"*Variables*" will attempt to explain the various uses of type declarations, saves, and variable definitions, "*Definitions*" tries to cover everything you need to know about writing your own general purpose definitions, "*Paths, pairs, pens and transforms*" tries to explain everything related to paths, pairs, pens and transforms, "*Conditions and loops*" is about making your program decide what to do next, and finally "*Colors and pictures*" is all about MetaPost output.

There is a lot of information in this set of articles, but this is *not* a manual. The actual user manual for MetaPost is `mpman.pdf` and is probably installed on your computer already as part of TEXLive. If you cannot find it, or if you do not have TEXLive at all: there is an only version at the TUG website at `https://www.tug.org/docs/metapost/mpman.pdf`. The point of this set of articles is not to replace that manual, but to elaborate and clarify some parts of it.

Happy MetaPost-ing!

Taco Hoekwater

Variables

Sparks, Tags, Suffixes and Subscripts

Abstract

MetaPost variables are rather complicated things. This article will attempt to explain the various uses of type declarations, `save`s, and `vardef`s.

Introduction

MetaPost inherits almost all of its syntax and internal structures from Metafont. Unfortunately, it did not inherit Metafont's documentation. The MetaPost manual by John Hobby (and later extended by the current development team) does a fine job of explaining how to make simple use of MetaPost and it explains where it differs from Metafont, but it is very light on details. The implicit assumption is that you should have read the Metafont book by D. E. Knuth and if you want to know more you should ask a Metafont guru for help.

That perhaps made sense in the nineties, but nowadays Metafont usage has dwindled to nothing (whereas MetaPost continues to be developed) and Metafont gurus are hard to find. The spiral-bound (cheap) version of the Metafont book is out of print, and even if you could find a copy of the Metafont book, it is not an easy read. In the late eighties when D. E. Knuth wrote Metafont, the terminology used for describing programming languages was quite different from what is popular today. Even back then, Metafont was an odd and very original language, with unfamiliar concepts. It all results in a book (and a programming language) that can only be used fully after a lot of careful study.

An immediately obvious weird and powerful thing about the Metafont and MetaPost languages is that the programs not only have the capability of solving linear equations, but also define a syntax to specify such equations in a partial, deconstructed format.

Less obvious is that Metafont and MetaPost variable names are quite special constructs.

In my opinion, the MetaPost manual as well as Hans Hagen's MetaFun manual do a fine job to explain linear equations but both authors are short on details when it comes to the syntax for identifiers (and variable names are an important part of such identifiers).

What follows is my attempt at explaining how variable names work in Metafont and MetaPost. There are other peculiarities of the programming languages that I could also write about, but I believe variable names are the most important things to explain for beginners.

Tokens, sparks and tags

As I assume all readers of this article are TEXies, I dare predict that you are familiar with the concepts of 'primitives' and 'macros'; if you did any kind of macro programming in TEX, you will also understand what a 'token' means to TEX.

While TEX interprets your input file, it converts the bytes it finds into tokens. Many tokens are just letters of text to be typeset. Some of the other tokens are primitive operations to TEX itself, like e.g. `\par`. Other tokens are macro names, like `\section`. The latter are in turn *expanded*, producing more tokens to be interpreted: text to be typeset, primitive operations, or perhaps other macros (that will be expanded in turn as well). In the end, your input converts itself into tokens that execute primitive operations of the typesetting engine.

Even if you are not so familiar with the intricacies of TEX, these are concepts common to any macro language and lexicographical analyzer and parser.

In TEX, (almost) all tokens are also commands for the engine. The only exceptions are things like space tokens to end number scanning and macro arguments that are stored for later use. Whenever you see a symbolic token like `\par`, you can be sure that it instructs TEX to do something at some point. In fact, for a word in a TEX paragraph, each individual character is a token that instructs TEX to typeset the glyph (an actual drawing of a particular character in a particular font) associated with it. And when TEX sees a digit in the right-hand side of an assignment like `\count0=123`, each separate digit from left to right instructs TEX to multiply the present value of `\count0` (starting at zero) by ten, and then successively add that digit.

TEX has only two types of tokens: ‘control sequence tokens’ and ‘character tokens’. Control sequence tokens are used for multi-letter constructs like primitive and macro names, and character tokens are used for everything else. Tokenization in TEX is controlled via the so-called `\catcode` or category code of the various input characters.

MetaPost has a somewhat different repertoire of tokens. There are ‘numeric tokens’ (floating point numbers), ‘string tokens’ (stuff between double quotes), and ‘symbolic tokens’ (everything else). Numeric and string tokens are quite straightforward and can be explained succinctly but symbolic tokens have to be explained in detail because they are quite different from control sequence tokens in TEX.

MetaPost does not have the `\catcode` command of TEX. However, it does have its own internal list of category codes, and those internal categories are used to construct tokens using a fairly short (but perhaps unexpected) list of rules.

When MetaPost is not in an exceptional situation like during the processing of `btex` …`etex` or `readfrom` where the standard MetaPost language conventions do not apply, it processes input text as follows:

If the next thing …

- ☐ is a space character, it is ignored;
- ☐ is a period character, it is ignored unless followed by another period or by a digit (see below for those);
- ☐ is a percent sign, everything further is ignored until an end of line character is seen;
- ☐ is a decimal digit or a period followed by a decimal digit, a numeric token is scanned and created;
- ☐ is a double (ascii) quote, a string token is scanned and created;
- ☐ is a left or right parenthesis, a comma, or a semicolon, a symbolic token is created with that value;
- ☐ is something not matched above, then it combines with the longest following sequence of characters in the same internal category as itself to become a single (multi-letter) symbolic token.

Creation of numeric tokens

Once the start of a numeric token has been detected, MetaPost runs a numeric token scanner that is specific to the current `numbersystem`. In the default `scaled` mode, a numeric token is the expected combination of digits and a dot. In the other `numbersystem` modes, an optional exponent can follow immediately afterwards. An exponent specification starts with the letter `e` or `E`, followed by an optional `+` or `-`, and then a series of digits. No intervening spaces are allowed (this limitation is present to ensure that the new syntax for numeric tokens has as small as possible an impact on existing MetaPost input).

Some examples of valid numeric tokens in all number systems are:

```
12
0.001
.2
```

The `double` and `decimal` number systems also allow numeric tokens to be created from this input:

```
12E0
1e-3
.000000002E8
```

Creation of string tokens

Once the start of a string token has been detected, MetaPost gobbles up characters from the current input line until it finds the matching double quote. The resulting string token consists of the letters in between those double quotes.

Creation of symbolic tokens

When MetaPost sees a left or right parenthesis, a comma, or a semicolon, it immediately creates a symbolic token with just that value.

The next rule in the list of processing actions mentioned above is what makes e.g. ‘`beginfig`’ be a single token. The actual internal categories (‘classes’, in MetaPost jargon) are defined by the list below, and they highlight some of the oddness of the MetaPost input language.

```
AZ _ az
< = > : |
‘ ’
+ -
/ * \
! ?
# & @ $
^ ~
[
]
{ }
.
```

For example, this nonsensical input:

```
beginfig.a ====>;
```

produces four symbolic tokens: ‘`beginfig`’, ‘`a`’, ‘`====>`’, and ‘`;`’. The period character and the space are ignored per the rules above.

Some things to meditate on

The statements above explain the existence of some fairly common MetaPost symbols such as '`beginfig`', '`:=`', '`..`' and '`---`'.

But it also means:

- that '`!?!`' and '`[[[[`' are valid symbols, which could be defined if you so desired;
- that there can never be symbolic tokens containing spaces, percent signs, double quotes, or digits;
- that period characters are often (but not always) equivalent to spaces (in fact, MetaPost usually replaces spaces with periods in log reports);
- that '`a.b.c`' is equivalent to '`a b c`';
- that numeric tokens are never negative (negative numbers are composed of two tokens);
- that string tokens are limited to a single line and never contain explicit double quotes (those strings need to be created using `char`).

Before reading on, make sure the above makes sense to you. Until you grasp these tokenization rules, you will be constantly surprised by what MetaPost thinks your input means.

Symbolic token processing

In MetaPost, it is not necessarily the case that a symbolic token is actually a command for the engine (as is the case for TEX-derived engines).

Symbolic tokens in MetaPost come in two possible types: those that are actually commands, and those that are not. To make it easier to talk about this distinction, the tokens that *do* signify commands are called **sparks**, and the ones that do not are called **tags**.

By definition, **sparks** are symbolic tokens that either refer to primitive operations (e.g. `:=`, `path`, and `withcolor`) or are defined to be macros (like `beginfig` and `fill`). Because those are the two groups of things that MetaPost considers 'commands'.

Symbolic tokens that do not refer to commands (**tags**) are the building blocks to construct variable names. All variable names are always constructed using *only* symbolic tokens that are **tags**, never **sparks** (also numeric tokens can be part of a variable name, but that will be covered later. For now, it is important to stress that **sparks** like `path` cannot be part of variable names).

In a simple assignment like

```
w := 12pt;
```

there are four symbolic tokens and one numeric token: `w`, `:=`, `12`, `pt`, and `;`.

The `w` and `pt` are **tags**. The other two symbolic tokens (`:=` and `;`) are (normally) **sparks**.

There was the word '(normally)' in the previous sentence. That is because like in TEX, MetaPost primitive operations are separate from the symbolic tokens that are normally used to execute them. There are options available in MetaPost to remap those connections, as will be explained in following sections.

Note that `pt` is actually a variable name. MetaPost does not have any built-in dimensions, so the typical `pt`, `cm`, … specifications are actually variables with a numeric value that are used as a multiplier for its native system, which is PostScript points. For example, the typesetting point `pt` is defined in the plain.mp macros as a numeric variable with the value 0.99626 (=72/72.27) as well as `cm` with the value 28.34646 (=72/2.54).

Variable names, suffixes and subscripts

Before getting into the actual details of variable names, I should explain some peculiarities of actual variable values in MetaPost.

In MetaPost, variables are always single objects: there are no arrays or dictionaries or objects or other compound variables in the language *at all*. Some language constructs may make it appear as if there are arrays and object structures, but MetaPost handles these constructs in a completely different manner from just about any other programming languages that you may be used to.

If you see `x1` in a MetaPost language definition, this refers to a variable that is actually called '`x1`'. It is *not* the entry at index '`1`' in the array '`x`'!

Variables in MetaPost are always strongly typed. That type comes from a fixed list of value types that are compiled into the binary and cannot be altered. The list of variable types is longer than average for a programming language: besides variable types for numerics and strings, MetaPost also has types for pairs (of numbers), paths, colors, et cetera. Those more complex variable values have components that can be queried and extracted separately.

For example, a variable valued as `pair` (the type normally used to express two-dimensional points) internally consists of two numerics that can be accessed using the `xpart` and `ypart` operations. While variable values are always single objects, that does not mean that they are always a single primitive value.

MetaPost deals with the lack of compound variables in a very interesting (or odd, depending on your viewpoint) way: variable names in MetaPost are not limited to a single symbolic token. Instead, variable names can be constructed using partial names (like firstnames and surnames, if you will).

The separate parts of a variable name can be either **tags** (as explained above) or numeric values.

A simple example is an equation like:

```
x1 = 12pt;
```

where the `x` and `1` are two parts that are actually combined into a single variable name.

So what exactly is a variable name, then? The parsing rules say that a variable name is built up from a **tag** optionally followed by a **suffix**. A **suffix** in turn is either a **subscript** or a **tag**, possibly followed by yet another **suffix**, and so forth. A **subscript** is either a numeric token, or a bracketed numeric expression (which then should result in a known numeric value).

There is no need to pre-declare *numeric* variables in MetaPost. Combined with the above parser rules means input like

```
x3ab c[2.1+1] f.4 = 12pt;
```

is perfectly valid. It defines a single variable with seven parts to its name: `x`, `3`, `ab`, `c`, `3.1`, `f`, and `.4`, having a numeric value of `11.95514` (12 times 0.99626). In 'normal' MetaPost jargon, it starts with a **tag** and has a **suffix** consisting of six parts. The first, fourth, and last **suffix**es are **subscript**s, and the other three are **tags**.

A **subscript** can be an immediate numeric token like `3` and `.4` in the above example, or it can be a bracketed expression like `[2.1+1]` that directly results in a numeric value. The brackets are required for MetaPost to interpret the **subscript** as an expression. Without them, the expression becomes part of the enclosing expression, which itself is usually an equation.

For example:

```
x3ab c 2.1+1 f. 4 = 12pt;
```

is also syntactically correct input (without brackets and with a space between `f.` and 4). However, it defines an equation for two variables:

```
x[3]ab.c[2.1] + 1*f[4] = 12pt;
```

This demonstrates that it is quite possible to write very obscure MetaPost code. To avoid confusing MetaPost (and yourself!) my advice is to always use square brackets around floating-point variable name segments, and to always use periods instead of blanks in between **tags**.

Numeric tokens cannot be negative, but the result of a numeric expression can be negative. The ability to use a numeric expression in a subscript is very powerful as it can contain calculus operations and even contain macro calls. The only requirement is that it has to produce a *known* numeric value. The following is allowed (although likely not very useful):

```
a[- floor uniformdeviate 20 + 5] = 12pt;
```

The parsing rules mean that

□ a string token can never be part of a variable name,
□ and neither can any **spark**,
□ and a variable name never starts with a numeric token or numeric expression.

The restriction on **sparks** in variable names is a cause of common errors in MetaPost input. Because numeric variables do not need to be predeclared in MetaPost, it is quite common to invent variable names on the fly. Chances are that at some point one of those spontaneous variable names uses a **spark** in some part of it, and an error will be reported by MetaPost.

When this happens, the actual error message will depend on the **spark**'s meaning, which can be quite confusing, indeed.

Declarations

Earlier it was mentioned that there is no need to pre-declare numeric variables. But numeric is not the only variable type that MetaPost knows about; the other types *do* need to be predeclared (otherwise they default to the numeric type).

In the simple cases, declarations look like this:

```
boolean   mybool;
cmykcolor mycolor;
color     mycolor;
numeric   mynumber;
pair      mypair;
path      mypath;
pen       mypen;
picture   mypic;
rgbcolor  mycolor;
string    mystring;
transform mytransform;
```

For a total of ten types (`color` is a pre-defined alias for `rgbcolor`).

While numeric variables do not need to be predeclared, the `numeric` keyword is still useful. That is because all declaration commands completely wipe out the current meaning of the to-be-declared object, whatever it is (as does `save`, to be described later).

For a slightly more complex case, you can declare multiple variables at the same time:

```
path p, q;
```

The argument to a declaration command is not exactly a variable name (or even a list of those), it is a bit more complicated than that: for starters, each element of the argument list is allowed to be a **spark**. Of course, after the declaration has been processed, any such **sparks** will become variable names (as they are now **tags**). This may be what you want, but it usually isn't. MetaPost does not give any warnings about redefining **sparks** in this way, so you have to be careful!

The statement

```
path path;
```

is allowed. It will be the last working path declaration in the current run, though, as it will turn `path` into a variable name as well as making the original meaning of `path` inaccessible.

Besides making sure you do not redeclare something important like `end` or `z`, also make sure not to have empty entries in the declaration list. If you do, the rule above will happily declare a variable for you whose name starts with a comma, in the process turning `,` into a **tag** and thus *breaking* every following statement that uses a comma anywhere in it!

The second big special thing about declaration lists is that they are not allowed to contain direct numeric tokens, and the only allowed bracketed numeric expressions are ones that are completely empty. This is because MetaPost insists that all variables whose names are identical except for subscript values have the same type.

You cannot have `a1` be a pair and `a2` be a color, for example (nor is this a very good idea from a code comprehension point-of-view). To enforce this rule, you can only use so-called ‘collective subscripts’, and the declaration would look like this:

```
pair a[];
```

After this, both `a1` and `a2` become unknown variables of type pair. To be more precise: all variables whose name consists of an initial **tag** `a` followed by a single **subscript** are now pairs.

If you are familiar with other programming languages, you may be tempted to look at the above example as an array declaration. But it is not: it just tells MetaPost that any variable with a combined name consisting of `a` followed by a numeric part will be of type `pair`. This does not prohibit you from using `a` *as if* it is an array, but it is important to realise that MetaPost does not actually see it that way.

Internally, **subscript** segments are stored as a linked list of numeric values in ascending order (the difference can be significant in terms of performance, especially for multi-dimensional pseudo-arrays).

An important advantage of how collective subscript declarations like the one above work is that it has *no* influence on any other variables whose names are not of the form `a` plus **subscript**. For example `a.colr` can still be a color, and if a pair `a.direction` pre-existed, then it will not have changed at all. Also, the variable `a` itself in not affected (and defaults to the numeric type unless declared otherwise). Even a nested set of variable names with each level having a different type is acceptable:

```
pair a;
path a[];
color a[]c;
```

Although, I would not necessarily recommend setups like this in actual use, as it gets confusing to yourself rather quickly.

A small warning: do not forget that the statements

```
path a.path;
color a.color;
```

are both illegal because they would result in variables names with **sparks** in them. You need something like this instead:

```
path a.pth;
color a.col;
```

Internal quantities

Besides user-defined variables, MetaPost also has a number of internal variables that are used by the MetaPost executable itself while processing your input. These are officially called 'internal quantities'. To keep things simple, all the names of the internal variable names are a single symbolic token.

Most have numeric type:

```
tracingtitles
tracingequations
tracingcapsules
tracingchoices
tracingspecs
tracingcommands
tracingrestores
tracingmacros
tracingoutput
tracingstats
tracinglostchars
tracingonline
year
month
day
time
hour
minute
charcode
charext
charwd
charht
chardp
charic
designsize
pausing
showstopping
fontmaking
linejoin
linecap
miterlimit
warningcheck
boundarychar
prologues
truecorners
defaultcolormodel
mpprocset
troffmode
```

```
restoreclipcolor
numberprecision
hppp
vppp
```

And a few have the string type:

```
outputtemplate
outputfilename
outputformat
outputformatoptions
jobname
numbersystem
```

It is not possible to change the type of these variables. If you try to do a type declaration anyway, you will end up with a new user-defined variable that happens to have the name of an internal quantity but is in fact not related to it at all.

From then on, the internal quantity has become inaccessible from within your code, even though the variable itself still exists. In situations where MetaPost needs to use that internal variable, it will use the value it held before you made it inaccessible.

There is a command to make new internal quantities: `newinternal`. Its usefulness is limited since proper variables can do a number of things that internal quantities cannot, but access to internal quantities is a little bit faster than normal variables, and that is even true for user-defined ones. On the other hand internal quantities can only receive *known* values. It can be quite useful to define new internal quantities for numerical constants.

For example, `plain.mp` defines `eps` as:

```
newinternal eps;
eps := .00049;
```

For the internal MetaPost parser, these internal quantity names pose a bit of a problem. Because they are variables, they are actually **tags**. However, internal quantities cannot be suffixed or subscripted. This means the definition of a variable as given earlier on is not quite correct. To be precise, a variable is either a single **tag** matching one of the currently known internal quantities, or it is the construct with a **tag** optionally followed by **suffix**es as explained earlier.

Save and interim

The `save` command functions in a very straightforward way: it processes a list of symbolic tokens (either **sparks** or **tags**), saves the current meaning or value in a safe place, and then converts the symbolic token into an undefined **tag**. It also makes every sub-variable that starts with that specific symbolic token be undefined. The `save` command operates on individual symbolic tokens, so it cannot be used to save just some sub-part of a segmented variable. It does not wipe-out and replace the previous variable as a new declaration would but instead, it makes the **tag** available locally.

The normal use for `save` is within a group starting with `begingroup` and ending with `endgroup`, like within `beginfig` …`endfig`.

The traditional `beginfig` macro contains the equivalent of

```
save x,y;
```

to make sure that any values of type `x[]` and `y[]` outside of the current figure do not have any undue influence, while still saving them for potential later use.

If you use save twice within a single group, it will actually do the saving and undefining two times. However, since both are unwound at the endgroup, whatever you saved first will always win out once you are outside of the group again.

For internal quantities, using save would not work, because the symbolic token becomes undefined and therefore unassignable. That is why there is a separate command for temporarily altering and internal quantity. The argument to interim looks like a normal assignment. The only difference is that the previous value is restored at the end of the group.

```
interim warningcheck := 0;
```

As with save, repeated calls do perform extra saves, but at the endgroup they are all unwound in save order, so the first saved value wins.

Let and def

A quick note on let and def for those of you that are familiar with their counterparts in TeX: while the principles are roughly equivalent in both languages, there are some important differences. MetaPost does not have user-controlled macro expansion, and it handles grouping in a completely different way, so the typical prefixes like \global and \expanded of TeX do not exist.

The let command makes one symbolic token be an alias for another symbolic token. It is typically used just before redefining a spark, but it can also be used to get more readable input. For example:

```
let graycolor = numeric ;
```

will improve readability of the input if you routinely want to defined specific greyscale colors. It is important to realise that graycolor is now a **spark**, because numeric is a **spark**.

The downside to let is that it only works (quite as you would expect) on **sparks**. The exact details are as follows:

- If the token on the left-hand side is a **tag** that starts a user-defined variable, then all variables that start with that **tag** become undefined (so besides redefining the token itself, it also destroys the whole variable structure);
- if the token on the right-hand side is a **tag** that starts a user-defined variable, then the left-hand side becomes undefined but the variable(s) on the right-hand side are left as-is;
- and if the token on the right-hand side is one of a set of currently defined delimiters, then the let will silently produce a bad delimiter definition (for matching delimiters there is a separate delimiters command).

But the exact details are not so important. The important thing to remember is that the let command is meant to provide aliases for **sparks** and cannot really be used for anything else other than that.

The def command is much more flexible. However, if you want to actually redefine a **spark** using def but still need the original meaning available somehow, then you have no choice but to first use let to store that original meaning in an alias.

As was implied earlier, def (and its cousins primarydef, secondarydef, and tertiarydef) produce **sparks** (since macro names are **sparks** according to the tokenization rules). This means that any name defined using a def command can no longer be used as part of the name of another variable. If that were allowed, it would be expanded immediately to its replacement text, and the macro's replacement text would be used instead of its name.

To elaborate, assuming there is a definition like

```
def up = (0,1) enddef;
```

then the variable name `a.up` would be impossible until that definition goes out of scope again, because MetaPost would actually interpret `a.up` as `a (0,1);` which then produces a syntax error (unless `a` is actually a macro itself that requires two delimited arguments).

Since this article is about variables and variable names, I do not want to delve into the details of the `def` command. But perhaps this would be a good topic for another article.

Variable definitions

The restriction that `def` always produces a **spark** is why there is a dedicated command for creating macros that are actually **tags**. This command is called `vardef`. In simple cases, the use of `vardef` is very similar to using `def`.

```
def stuff =
  fill unitsquare
enddef;
```

and

```
vardef stuff =
  fill unitsquare
enddef;
```

appear equivalent when they are executed. But there is a difference in execution: the `vardef` version actually expands into:

```
begingroup
  fill unitsquare
endgroup
```

The extra grouping makes the macro expansion syntactically equivalent to a variable when the MetaPost needs to see an expression next. This is important because it avoids confusing the MetaPost parser.

This works because grouping in MetaPost is a bit unusual (yet another way in which MetaPost is unusual!) in that the `begingroup` ...`endgroup` block is not only seen as a list of statements grouped together. It can also be used as an expression. When viewed as an expression (which is usually the case for `vardef` macro expansions), all the statements in the group are executed as normal, but the last expression inside the group (it could be empty) is taken as the value to use for the expression outside of the group. It is precisely this oddity of grouping that makes `vardef` definitions syntactically equivalent to variables.

Incidentally, it also makes grouping behave similar to an anonymous function call with one return value.

The extra grouping usually will not matter, but it means you cannot do things like

```
stuff withcolor green;
```

which makes sense once you realise that `vardef` is supposed to equate to a variable. If we assume for a moment that there was instead a normal path variable named `stuff`, then the call would look like this:

```
fill stuff withcolor green;
```

and indeed, after adjusting the `vardef` to

```
vardef stuff =
  unitsquare % earlier 'fill' deleted
enddef;
```

it works just fine.

In some cases, the implicit extra grouping is an impediment, and it would be better to use def. But sometimes that extra grouping level can be a bonus: it allows trivial macro definitions that need grouping to be a bit shorter. Still, this is only a very minor advantage, and the MetaPost manual explicitly warns against abusing vardef just for grouping.

So why is vardef useful?

First, because vardef defines a new **tag** instead of a **spark**, the symbolic token itself can still be used in the middle of an unrelated compound variable name. Occasionally, you may want to define a macro with a name that would also make sense as a suffix to another variable. The Metafont book highlights the example of dir. The variable macro dir is defined as a vardef precisely because doing it this way means it is still legal to have a pair variable named p5dir.

Also, because vardef produces a **tag**, it can be used to create variable 'names' that are actually macros. This is not just the case in standalone situations like with dir. Macros that are vardef'ed can also be used at the end of compound variable names. For example, you could have:

```
rgbcolor p[]col;
vardef p[]dir=
  (#@dx,#@dy)
enddef;
p5col = red;
p5dir = up;
```

and that vardef definition would not interfere with the rgbcolor declaration (see below for the usage of the special #@ token).

There is a more specialized use of vardef as well. The heading of a vardef allows a special syntax that is a little more elaborate than a normal def. This is easiest to explain with an example from plain.mp:

```
vardef z@#=
  (x@#,y@#)
enddef;
```

This defines the variable macro z. What makes this definition heading of z special is that the definition now has a built-in parameter of type **suffix** that is named @# (remember that @# is a single token, as explained in the tokenization rules at the start of this article). The use of @# in the definition heading triggers this behaviour. You can always ask for @# in the replacement body, but if @# was not also used in the heading, @# would always be empty.

There is a subtle difference between this definition of z and the more naïve version:

```
vardef z suffix v =
  (x.v,y.v)
enddef;
```

The special token @# only applies to a subsequent suffix; the suffix that becomes the argument may not be enclosed in parentheses (unlike in the second definition, where parentheses when calling are optional). Getting into the details of these definition headings is quite far outside of the scope of this article but for advanced usage with multiple arguments to the vardef, the main advantage of @# is that when the vardef is called, it allows for an undelimited suffix that is processed before any other arguments are considered. With ‘normal’ definition headings, this is impossible to do.

I had an example of usage here in an earlier draft, but that created more confusion than it solved because definition headings needed explaining in detail. Just remember that the special token @# in a vardef definition heading makes it especially useful for manipulating sub-variables (like the actual z definition from plain.mp does).

Finally every vardef, with or without the special @#, also has two other special implicit arguments that can be used anywhere in the replacement text. The special argument name @ returns the last part of the name of the defined macro, and the special argument name #@ returns the complement: all the parts before the last one.

When is this useful? Look at this:

```
vardef p[]dir=
  (#@dx,#@dy)
enddef;
```

After this definition, p5dir expands into:

```
(p5dx,p5dy)
```

allowing you to write, for example:

```
p5dir = up;
```

to define the dx and dy subvariables, and query those values by

```
if p5dir = up: .... fi
```

which looks and feels a lot nicer than having to manipulate the dx and dy variables ‘manually’ like so:

```
(p5dx,p5dy) = (0,1);
```

In definitions like p[]dir, the special token @ which expands into the macro ‘name’ is not very useful (we already know that it is dir), but keep in mind that **subscript**s can also be vardef macros themselves. Since @ expands into the actual subscript in that case, it can then be used to differentiate between macro calls for specific subscripts by using a numerical comparison, like this:

```
vardef a[] =
  if odd @: message("odd")
  else:     message("even")
  fi
enddef;
a1;  % prints "odd"
a20; % prints "even"
end.
```

In cases where one of the special tokens is not guaranteed to be a **subscript**, to test its value you could use the str command instead (this makes most sense with implicit suffixes):

```
vardef a@# =
  if str @# = "o": message("odd")
  else:            message("even")
  fi
enddef;
a.o; % prints "odd"
a.e; % prints "even"
```

A warning about using `vardef`: because `vardef` is a macro, it only works as the *last* part in a complete variable name. After the `p[]dir` definition above, you can **not** now add another suffix to create a new variable name:

```
pair p[]dir.target; % WRONG!
```

This is disallowed, because that set of variables would actually be inaccessible.

Because of how the MetaPost parser works, the `target` part of this name would *need* to become a suffix argument to the `p[]dir` macro for the syntax to be correct. But in this case, as the macro is defined without a suffix argument, it is never picked up, and the result is a syntax error:

```
! Isolated expression.
<to be read again>
                   target
```

If you really want to write things like `p5dir.target` in the input, you could extend the definition of `p[]dir` to also accept the undelimited suffix `@#`, and then process the `target` within the macro expansion but note that `p5dir.target` would then not a variable name. The variable name is `p5dir`, with the special type `vardef`, and it receives the argument suffix `target`.

Acknowledgements

The author of this article had been the MetaPost maintainer for quite a while and even ported the MetaPost source code from the PascalWEB source to CWEB. Even so, this text could not have been written as quickly and thoroughly without the invaluable aid of the two main sources explaining the language: the *Metafont book* by D. E. Knuth, and *MetaPost – a user's manual* by John D. Hobby, the creator of MetaPost. If you want to become more fluent in MetaPost, then of those two in particular the *Metafont book* is a must-read. While MetaPost extends the Metafont language in some areas (like the addition of color) and removes some other areas (especially the underlying bitmap model), the core principles of the language as well as its syntax rules have remained essentially the same as in Metafont.

Special thanks also go to my proofreaders Alan Braslau, Hans Hagen and Gerben Wierda. Their proposed improvements have resulted in a much better article than I would have been able to create on my own.

Taco Hoekwater

Definitions

Abstract

Definitions in MetaPost are a fairly complicated subject. This article tries to cover everything you need to know about writing your own definitions, but it assumes a fair bit of familiarity with MetaPost's data types and general syntax. In particular, I assume you have read the preceding 'Sparks, Tags, Suffixes and Subscripts' article.

Macro definitions

The def command defines a token to be replaced with a replacement text. This is close to how macros work in TEX, and is therefore the easiest to explain. So let's start with this.

In its simplest form, it looks like this:

```
def ⟨symbolic token⟩ =
    ⟨replacement text⟩
enddef;
```

Since each def is a complete statement in MetaPost, the semicolon after enddef is required. There is no need for a semicolon to end the ⟨replacement text⟩, because the expansion of the macro can happen in the middle of an expression, where an extra semicolon would interfere. The ⟨replacement text⟩ itself can be almost anything (with some minor limitations, see section for the exact rules).

A simple predefined example is the -- macro that is used to draw straight lines in path definitions. It is defined like this:

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

Macros are much more useful if they take arguments. Here is where MetaPost is quite different from TEX (or any other language, for that matter). There are quite a few ways to write the definition such that the defined macro accepts arguments in some form or another. There is an option for one undelimited argument and/or multiple delimited arguments, all of which come in various types.

Let's discuss the three basic types of arguments first: These are expr, suffix, and text.

expr	arguments are for passing expression values.
suffix	arguments are for passing (parts of) variable names.
text	arguments are for passing a list of arbitrary tokens.

In the most simple form, defining a macro with an argument looks like:

```
def mymac (expr a) =
...
```

where you can replace expr with suffix or text. We will talk about passing multiple arguments later on.

The parameter name a in this example is actually a ⟨symbolic token⟩ itself. It has to be a single symbolic token (not a literal number or a string), but it is not required to be alphanumeric. Any symbolic token will do, except for tokens explicitly set as outer.

You could do:

```
def mymac (expr ,) =
  , = 5
enddef;
```

although that is only useful in obfuscation contests. What you cannot do is use numeric suffixes like in many other languages:

```
def mymac (expr a1) =  % Error
  a1 = 5
enddef;
```

The parameter names do not actually exist as variables; they are only there to give you something to use in the ⟨replacement text⟩. Whatever the value of the parameter is, it is stored in a temporary value slot that has no name (these things are called `capsules` internally). You can actually see these slots in the terminal or log if you turn on tracing. In the trace output they are represented as the parameter type in uppercase with a sequence number attached, for example:

```
tracingall;
def mymac (expr a) =
  a = 5
enddef;

mymac(5);
```

will show:

```
mymac(EXPR0)->(EXPR0)=5
(EXPR0)<-b
{(b)=(5)}
## b=5
```

While there is not really a parameter named `a`, the use of `a` as a placeholder for the `capsules` does make it seemingly impossible to refer to an actual variable or command named `a`. But there is a solution to that.

If you want to access an outside name `a` within a macro that has a parameter named `a`, you can still do so by using the `quote` command:

```
def mymac (expr a) =
  quote a = a
enddef;

mymac(b);
```

The first ('quoted') `a` is now referring to an outside variable. In this case, it will set up an equation `a = b` because `b` is the replacement value of the parameter `a`.

Using `quote` like this works for all three parameter types, and it can also help you if you happen to have a macro parameter name that matches a MetaPost command's name. However, a small warning: it is usually better *not* to write macros that alter outside variables as side effects because you are likely to confuse yourself. And for access to other macros and/or commands, it is much better to come up with unique parameter names.

expr arguments

Arguments of type `expr` pass a value to the macro. The argument has to be a valid expression, and that expression is interpreted to produce a value that is then stored in the temporary variable that is used in the replacement text of the macro.

The expression is interpreted as far as possible first. Here is an example: If the macro mymac is defined to have an argument of type expr, you could call it like this:

```
mymac(5);
```

or like this:

```
mymac(2+3);
```

and in both cases the replacement text will see the value 5. But the value does not have to be 'known'. In these two calls:

```
mymac(5b);
```

```
mymac((2+1)*b +2b);
```

the replacement text will see 5b if the variable b is not known at the time of calling.

Because the replacement text works with a nameless variable's value, it is not assignable. This means that inside the replacement text, the formal names of expr parameters cannot be used on the left side of an assignment (:=). For example, this is forbidden:

```
def mymac (expr a) =
  a := 5 % Error
enddef;
```

But that does not mean you cannot alter the value itself, because equations (=) with that parameter name are still allowed:

```
def mymac (expr a) =
  a = 5
enddef;
```

```
mymac(5b);
```

is correct input and resolves the outer variable b to the integer value 1 (its value will remain known even after the macro call).

Be aware, though, that macros that have such hidden side-effects are hard to maintain, so you need to be quite certain of how the macro will be called if you make use of this. For general purpose macros, it is almost always better to receive and/or return a value instead of modifying the parameter. The above definition would be cleaner if written like this:

```
def mymac(expr a) =
  a
enddef;
```

```
5b = mymac(5);
```

Well, this is a silly example, of course, but the point should be clear, I hope.

suffix arguments

Arguments of type suffix pass a 'suffix' to the macro. A 'suffix' is the trailing part of a variable name, possibly consisting of multiple segments, and possibly being the whole name. The argument passed to the macro really is the (partial) name of a variable. Per the normal rules, if you try to pass an undefined suffix (or whole variable), it is initialized to be of type numeric.

```
def mymac(suffix a) =
  a = 5
enddef;
```

```
mymac(b);
```

assigns the value of 5 to the variable named b, assuming it is of type numeric. If it is not numeric, an error will be raised.

And:

```
def mymac(suffix a) =
  pair a
enddef;

mymac(b);
```

converts b into a variable of type pair.

Since this is all resolved by name, this:

```
def mymac(suffix a) =
  pair c.a
enddef;

mymac(b);
```

sets up c.b to be of type pair. The variable c itself remains untouched, just as if you wrote

```
pair c.b;
```

without any macro definition.

Macros with suffix parameters can sometimes be a little complicated to read because of the 'passing a name as a parameter value' rule. Just to be clear, inside the macro there is at *no* time a variable named c.a. In fact, if there was a variable c.a defined before the macro call, then it will remain completely untouched.

Another effect of the suffix parameter passing a name instead of a value is that it actually *can* be used on the left side of an assignment:

```
def mymac(suffix a) =
  a := 5
enddef;

mymac(b);
```

does indeed assign the value 5 to the variable b.

text arguments

Arguments of text pass literal input text as a macro argument. That text fragment is not even limited to a single expression or statement.

```
def mymac (text a) =
    a = 5
enddef;

mymac(b);
```

Of all three possible argument types, this is the closest to a 'true' macro replacement. The argument's input text is processed exactly where it is called in the replacement text:

```
def mymac (text a) =
    c = 5;
    a = c
enddef;

mymac(b);
```

This gives b the value of 5 as well as doing the same to c. This is another case where it is very important to remember that there is never a variable named a. The parameter name is just a placeholder that temporarily shields any preexisting variable named a.

Because of the rules for text parameters, this is also allowed:

```
def mymac (text a) =
    c = a c
enddef;

mymac(5; a =);
```

But constructions like this are not advised if you want your code to remain understandable.

So how does MetaPost decide when a text argument to macro has ended? When it sees an unmatched closing parenthesis.

Matching parentheses in the argument are counted, so

```
def mymac (text a) =
    a + 5
enddef;

mymac(b = (3 + 4));
```

works just fine and equates b to 12. You cannot get an unmatched parenthesis into the replacement text without some trickery (by defining a macro with a single parenthesis as its replacement text and using that in the macro call instead of a literal ().

Multiple delimited arguments

So far, we have only dealt with single arguments, but it is also possible to have multiple arguments.

The formal syntax definition for delimited arguments is as follows:

```
def ⟨symbolic token⟩ ⟨delimited part⟩ =
    ⟨replacement text⟩
enddef;
⟨delimited part⟩ → ⟨empty⟩
    | ⟨delimited part⟩ (⟨parameter type⟩ ⟨parameter tokens⟩)
⟨parameter type⟩ → expr | suffix | text
⟨parameter tokens⟩ → ⟨symbolic token⟩ | ⟨parameter tokens⟩, ⟨symbolic token⟩
```

Reading a formal syntax like the one above takes a bit of practice, but converted to English it says that the delimited part of a macro definition header is possibly an empty sequence of items enclosed in parentheses. Each of these parenthesized sequences start with expr, suffix or text followed by a comma-separated list of at least one symbolic token.

Not expressed in the formal syntax is that whitespace is ignored except as a way to separate tokens, as is normal in the MetaPost language.

Starting with some examples to illustrate the above syntax rules will hopefully help you learn how to apply these rules. Here are some correct ways to start a definition:

```
def mymac =
def mymac(expr a) =
def mymac(expr a,b) =
def mymac(expr a)(expr b) =
def mymac(suffix a)(expr b, c) =
def mymac(suffix a)(expr b, c)(suffix d)(text e) =
```

When a macro that is defined with multiple delimited arguments is called, specifying the internal delimiters is optional unless the argument is of `text` (this will be explained below). You can even insert delimiters that were not there in the definition or split the groups for `expr` and `suffix` parameters differently.

That last `mymac` macro above with the five delimited arguments in four groups can be called in various ways:

```
mymac (a,b,c,d,e);
mymac (a)(b)(c)(d)(e);
mymac (a,b)(c,d,e);
```

But all five arguments are required, and all of them must be part of delimited group. Here are some attempts that are *not* allowed:

```
mymac (a,b);          % bad 1
mymac a;              % bad 1
mymac (a)(b)()()(e);  % bad 2
mymac (a,b,,,e);      % bad 2
mymac (a,b)c(d)(e);   % bad 3
mymac (a,b)(c,d) e;   % bad 3
```

The ones marked `bad 1` are obviously illegal because some parameters are missing completely.

The ones marked `bad 2` are illegal because parameters cannot be empty. If you want to implement some sort of default behaviour, you will have to pass a variable of a special type or value, and deal with that as a special case in the replacement text. Just skipping the parameter is not allowed.

The ones marked `bad 3` are disallowed because all delimited arguments must be delimited.

But the rules above do not mean that you have to always specify all five arguments explicitly. MetaPost expands macros as it searches for the opening delimiters of the arguments of a macro call, so this is legal input:

```
def helper = (d)(e) enddef;
mymac (a,b)(c) helper;
```

You could even put all of the delimited arguments in separate macro definitions.

Coming back to that last `bad 3` case for a bit, this is allowed:

```
def e = (f) enddef;
mymac (a,b)(c,d) e;
```

Here, the macro `e` passes the replacement text of itself as the final argument to `mymac`.

But this is also possible:

```
def e = (f) enddef;
mymac (a,b)(c,d)(e);
```

And here, the macro `e` *itself* is the final argument to `mymac`. That is because once MetaPost has found a symbolic token that will become a macro argument, it will not expand it any further, so the macro itself is passed as the `text` argument instead of its replacement.

You need to remain aware of the fact that the expansion of macros only happens while MetaPost is actively looking for argument delimiters (opening parentheses and commas). You could do this:

```
def e = (f enddef;
mymac (a,b)(c,d) e);
```

or this:

```
def e = ,f enddef;
mymac (a,b)(c,d e);
```

or even this:

```
def c = f) enddef;
mymac (a,b)(c(d,e);
```

but not this:

```
def e = f) enddef;
mymac (a,b)(c,d)(e;
```

because in the last example, MetaPost has already stopped looking for more arguments. It knows that there are only five arguments, so it does not bother to scan for a delimiter that would start a sixth argument.

Multiple and `text` *arguments* Because of the nature of `text` arguments, they need an extra rule. It is possible to define a delimited macro with multiple text arguments like this:

```
def mymac (text e,f) =
    show e; show f;
enddef;
```

But this macro cannot be called without extra parentheses. With:

```
mymac(g,h);
```

the replacement text of the `e` argument becomes `g,h` and MetaPost stops with an error about the missing argument `f`. If there are multiple `text` arguments or other arguments following a `text` argument, extra parentheses groups are required. This is OK:

```
mymac (g)(h);
```

and this is also ok:

```
mymac (g; i; j; k; l)(h);
```

There is a simple rule to remember: always put text arguments in separate parentheses.

Undelimited arguments

Besides delimited arguments, macros can also have one undelimited argument. There can be only one of these and it has to be the last argument, but all three types are allowed, and there are some extra options as well. The syntax for undelimited arguments is as follows:

```
def ⟨symbolic token⟩ ⟨delimited part⟩ ⟨undelimited part⟩ =
   ⟨replacement text⟩
enddef;
⟨undelimited part⟩ → ⟨empty⟩
      | ⟨parameter type⟩ ⟨parameter⟩
      | ⟨precedence level⟩ ⟨parameter⟩
      | expr ⟨parameter⟩ of ⟨parameter⟩
⟨precedence level⟩ → primary | secondary | tertiary
```

(The ⟨delimited part⟩, ⟨parameter type⟩ and ⟨parameter⟩ have not changed and are omitted from the listing for brevity).

The three types of argument we have already discussed in the previous paragraph are the familiar cases. They are much like their delimited counterparts, except without delimiters. But there are a few extra notes:

□ An `expr` argument grabs the longest expression it can find. When such a macro is called, MetaPost also allows an `=` or `:=` just before the argument.
□ A `suffix` argument takes the longest suffix it can find. MetaPost allows that suffix to be enclosed in parentheses.
□ A `text` argument stops at the next semicolon or `endgroup`.

The new options are:

□ `primary`, `secondary` and `tertiary` arguments are just like `expr`, except they grab a 'smaller' argument (a partial expression). This will be explained below.
□ `expr` ⟨parameter⟩ `of` ⟨parameter⟩ is useful for creating macros that mimic the primitive operation `point t of p`. It grabs the longest syntactically correct ⟨expression⟩ `of` ⟨primary⟩ (see page 26 for the explanation of ⟨expression⟩ and ⟨primary⟩). It is not possible to fake the `point of` primitive syntax in another way.

Operator definitions

Quite often, you will want a macro defined with an `expr` argument to take only a part of the following expression instead of the whole of it. This is where the `primary`, `secondary` and `tertiary` keywords come in, as they operate on a *part* of an expression.

But for a better understanding, we need to back up a bit. Just like there are syntactic rules for macro definitions, there are formal rules for all other bits of MetaPost programs as well.

A MetaPost program is a sequence of statements. Most statements are internal commands, equations, or assignments. Expressions are part of equations and assignments. And expressions can be further subdivided into operators that work on variables or on further subdivisions of expressions.

There are a few other options for statements, and all the expression cases exist for all variable types (booleans, numerics, pairs, etc.). For brevity, I will concentrate on the numeric expressions to explain what is going on, and ignore all those other cases. In the syntax definition below, all ⟨...⟩ are extra rules that I have skipped.

Here are the parts that are relevant right now:

```
⟨equation⟩ → ⟨expression⟩ = ⟨right-hand side⟩
⟨assignment⟩ → ⟨variable⟩ := ⟨right-hand side⟩
⟨right-hand side⟩ → ⟨expression⟩ | ⟨...⟩
⟨expression⟩ → ⟨numeric expression⟩ | ⟨...⟩
⟨numeric expression⟩ → ⟨numeric tertiary⟩
⟨numeric tertiary⟩ → ⟨numeric secondary⟩
        | ⟨numeric tertiary⟩ + | - ⟨numeric secondary⟩
        | ⟨...⟩
⟨numeric secondary⟩ → ⟨numeric primary⟩
        | ⟨numeric secondary⟩ * | / ⟨numeric primary⟩
⟨numeric primary⟩ = ⟨numeric atom⟩
        | ⟨numeric atom⟩ [ ⟨numeric expression⟩ , ⟨numeric expression⟩ ]
        | ⟨...⟩
⟨numeric atom⟩ → ⟨numeric token⟩
        | ( ⟨numeric expression⟩ )
        | ⟨...⟩
```

Working top-down, you can split a numeric expression into parts to the left and right of a plus or minus operation. Those left and right sides can each be split further into left and right sides of the multiply and divide operators. These sides can each be split even further into the arguments of the mediation operator.

In case you are wondering: the off-by-one between the left and the right is what makes operators in MetaPost left-associative.

Following these rules, let us investigate this expression:

```
4*(a+1) - b / 2[4,8]
```

Using the nomenclature from the official syntax, we can say that there are four primaries: `4`, `(a+1)`, `b` and `2[4,8]`. The two secondaries are `4*(a+1)` and `b / 2[4,8]`. The single tertiary is the whole `4*(a+1) - b / 2[4,8]`, which is also the whole expression.

The content of `(a+1)` is itself a nested expression, which can be subdivided using the same rules, but with a few shortcuts: `a` and `1` are the primaries. These are also the numeric secondaries, because there are no multiplication or division operations specified. The tertiary is `a+1`, which is also the expression value.

MetaPost supports four levels of operators: primary, secondary, tertiary, and expression. Not all value types have operators defined for all levels, though. That is why a ⟨numeric expression⟩ is the same as a ⟨numeric tertiary⟩. The rules for ⟨string expression⟩ look quite different:

```
⟨string expression⟩ → ⟨string tertiary⟩
        | ⟨string expression⟩ & ⟨string tertiary⟩
⟨string tertiary⟩ → ⟨string secondary⟩
⟨string secondary⟩ → ⟨string primary⟩
⟨string primary⟩ → ⟨string variable⟩
        | char ⟨numeric primary⟩
        | ⟨...⟩
```

As you can see, strings only have operators on the primary and expression level. The operators for the other types are yet again different, but the expression structure stays the same.

When you are planning on defining operators yourself, it would be helpful to have a list of the current operators and their level. But alas, such a list typically does not exist because the built-in operators that are part of the bare MetaPost binary itself can (and usually will be) augmented by the MetaPost macro package you are using. If you are lucky, the macro package manual contains a concise list somewhere. If not, you will have to do some trial and error until your definitions 'work' ...

Unary operator definitions

Getting back to macro definitions: `expr` grabs an ⟨... expression⟩. `primary` grabs a ⟨... primary⟩, `secondary` grabs a ⟨... secondary⟩ and `tertiary` grabs a ⟨... tertiary⟩.

It should now be clear that in:

```
def mymac primary arg =
 arg
enddef;
res = mymac 4*(a+1) - b / 2[4,8];
```

the argument is the `4`.

In this version:

```
def mymac secondary arg =
 arg
enddef;
res = mymac 4*(a+1) - b / 2[4,8];
```

the argument is the `4*(a+1)` (well, actually it is `4a+4`, because MetaPost interprets the partial expression before storing it in the parameter capsule).

And:

```
def mymac expr arg =
 arg
enddef;
res = mymac 4*(a+1) - b / 2[4,8];
```

and:

```
def mymac tertiary arg =
 arg
enddef;
res = mymac 4*(a+1) - b / 2[4,8];
```

both get the full expression as argument (actually `-0.08333b+4a+4`).

The net effect of using an undelimited `expr`, `primary`, `secondary` or `tertiary` is that you have created a new unary operator at that level. See section for how to define binary operators for the top three levels. Primary operators in MetaPost are always unary operators.

Binary operator definitions

It is now time to learn about binary operator definitions.

⟨macro definition⟩ → ⟨macro heading⟩ = ⟨replacement text⟩ `enddef`
⟨macro heading⟩ → `primarydef` ⟨parameter⟩ ⟨symbolic token⟩ ⟨parameter⟩
  | `secondarydef` ⟨parameter⟩ ⟨symbolic token⟩ ⟨parameter⟩
  | `tertiarydef` ⟨parameter⟩ ⟨symbolic token⟩ ⟨parameter⟩

A macro defined using `primarydef` defines a new operator with a ⟨... secondary⟩ on the left and a ⟨.. primary⟩ on the right of its name. For a `secondarydef` that is a ⟨... tertiary⟩ on the left and a ⟨.. secondary⟩ on the right, and for a `tertiarydef` it is an ⟨... expression⟩ on the left and a ⟨... tertiary⟩ on the right.

This definition creates an alias for `*`:

```
primarydef a mult b =
  a * b
enddef;
```

The names of the primitives seem off by one compared to the keywords for undelimited `def` arguments that we encountered earlier. But since MetaPost does not support binary primary operators, there would be only three possible levels anyway. You'll just have to get used to that. And at least `tertiarydef` sounds more natural than the fictitious '`exprdef`'. And remember, you can define unary binary operators with an undelimited `def` argument of type `primary`.

Variable definitions

Note: much of this section is copied and modified from my earlier paper.

The previous commands `def`, `primarydef`, `secondarydef`, and `tertiarydef` have one thing in common: they produce what MetaPost calls **spark**s. Effectively, you are defining a new ‘command’ instead of a ‘variable’. But sometimes you may want a macro to behave more like a named variable. For that to work, the macro has to be what MetaPost calls a **tag**.

The restriction of `def` always producing a **spark** is why there is a dedicated command for creating macros that are actually **tag**s. That command is `vardef`.

Because `vardef` defines a new **tag** instead of a **spark**, the name that is being defined can still can be used in the middle of an unrelated compound variable name. Occasionally, you may want to define a macro with a name that would also make sense as a suffix to another variable. The Metafont book highlights the example of `dir`. The variable macro `dir` is defined as a `vardef` precisely because doing it that way means it is still legal to have a pair variable named `p5dir`.

In simple uses, use of `vardef` is very similar to using `def`.

```
def stuff =
  fill unitsquare
enddef;
```

and

```
vardef stuff =
  fill unitsquare
enddef;
```

appear equivalent when they are executed. But there is a difference in execution. The `vardef` version actually expands into:

```
begingroup
  fill unitsquare
endgroup
```

The added grouping makes the macro expansion syntactically equivalent to an expression, which is important because it avoids confusing the MetaPost parser. We will get to the use of grouping later on.

Here is the formal definition of the syntax of `vardef`:

$\langle$macro definition$\rangle$ $\rightarrow$ $\langle$macro heading$\rangle$ = $\langle$replacement text$\rangle$ `enddef`
$\langle$macro heading$\rangle$ $\rightarrow$ `vardef` $\langle$declared variable$\rangle$ $\langle$delimited part$\rangle$ $\langle$undelimited part$\rangle$
  | `vardef` $\langle$declared variable$\rangle$ `@#` $\langle$delimited part$\rangle$ $\langle$undelimited part$\rangle$

The $\langle$delimited part$\rangle$ and $\langle$undelimited part$\rangle$ are the same as before and are not repeated.

The use of $\langle$declared variable$\rangle$ instead of the $\langle$symbolic token$\rangle$ from the earlier definition commands is important: This is what makes this type of definition produce a **tag** instead of a **spark**. The $\langle$declared variable$\rangle$ is actually the syntax rule for a single item in a type declaration command (`boolean`, `path`, `picture`, etc.).

You can define segmented variable names, and even use collective subscripts:

```
vardef mymac[]arr =
  4
enddef;
```

defines all variables of the form `mymac[]arr` to be macros that expand into `begingroup 4 endgroup`.

The second option for the $\langle$macro heading$\rangle$ of a `vardef` syntax introduces an extra keyword `@#`. This is easiest to explain with an example from `plain.mp`:

```
vardef z@#=
  (x@#,y@#)
enddef;
```

This defines the variable macro `z`. What makes this definition of `z` special is that it now has a built-in parameter of type ⟨suffix⟩ that is named `@#`.

There is a subtle difference between this definition of `z` and the more naïve version:

```
vardef z suffix v =
  (x.v,y.v)
enddef;
```

The special token `@#` only applies to a subsequent suffix; the suffix that becomes the argument may not be enclosed in parentheses, unlike in the definition with an undelimited argument. This makes the special definition exceptionally useful for manipulating sub-variables (like `z` does).

The `@#` somewhat replaces `suffix v`. You can still define a macro like this:

```
vardef mymac @# suffix v =
  (x@#v,y@#v)
enddef;
```

but you always have to call that macro with parentheses around parameter `v`, otherwise the whole argument becomes part of the `@#` suffix:

```
origin = mymac1right;
```

will have `1right` as `@#` and `v` empty. With

```
origin = mymac1(right);
```

this does not happen, but then you could have equivalently defined `mymac` as

```
vardef mymac @# (suffix v) =
  (x@#v,y@#v)
enddef;
```

Finally, every `vardef`, with or without the special `@#`, also has two other special implicit arguments that can be used anywhere in the ⟨replacement text⟩. The special argument name `@` returns the last segment of the name of the defined macro itself, and the special argument name `#@` returns the complement: all segments before that last one.

When is this useful? Look at this:

```
vardef p[]dir=
  (#@dx,#@dy)
enddef;
```

After this definition, `p5dir` expands into:

```
(p5dx,p5dy)
```

allowing you to write, for example:

```
p5dir = up;
```

to define the `dx` and `dy` subvariables, and query those values by

```
if p5dir = up: .... fi
```

which looks and feels a lot nicer than manipulating the `dx` and `dy` variables ‘manually’.

In definitions like `p[]dir`, the special token `@`, which expands into the macro ‘name’, is not very useful (we already know that it is `dir`), but keep in mind that **subscript**s can also be `vardef` macros themselves. Since `@` expands into the actual subscript in that case, it can then be used to differentiate between macro calls for specific subscripts by using a numerical comparison, like this:

```
vardef a[] =
  if odd @: message("odd")
  else:     message("even")
  fi
enddef;
a1;  % prints "odd"
a20; % prints "even"
end.
```

In cases where the expansion of one of the special tokens (`#@`, `@`, or `@#`) is not known to be numeric beforehand, to test its value, you can use the `str` command instead to force an expression with type ⟨string⟩ (this makes most sense with implicit suffixes):

```
vardef a@# =
  if str @# = "o": message("odd")
  else:            message("even")
  fi
enddef;
a.o; % prints "odd"
a.e; % prints "even"
```

A warning about using `vardef`: because the result of the `vardef` is a macro, it only works as the *last* typed segment in a complete variable name. After the definition above, you can **not** now add another suffix:

```
pair p[]dir.target; % WRONG!
```

This is disallowed because that set of variables would actually be inaccessible.

Because of how the MetaPost parser works, the `target` part of the name would always become a suffix argument to the `p[]dir` macro. In this case, as the macro is defined without a suffix argument, the result would be a syntax error. However, if you really want to write things like `p5dir.target`, you could extend the definition of `p[]dir` to also accept the undelimited suffix `@#`, and then process the `target` within the macro expansion.

In some cases, the implicit extra grouping added by `vardef` is an impediment, and it would be better to use `def`. But sometimes that extra grouping level can be a bonus as well: it allows trivial macro definitions that need that grouping to be a bit shorter. Still, that is only a very minor advantage, and the MetaPost manual explicitly warns against abusing `vardef` just for grouping.

Grouping

The sequence `begingroup` ...`endgroup` can be used as a standalone statement. The formal definition of ⟨statement⟩ looks like this:

$$\begin{aligned}\langle\text{statement}\rangle \;\rightarrow\; & \langle\text{equation}\rangle \mid \langle\text{assignment}\rangle \mid \langle\text{declaration}\rangle \\ \mid\; & \langle\text{definition}\rangle \mid \langle\text{title}\rangle \mid \langle\text{command}\rangle \mid \langle\text{empty}\rangle \\ \mid\; & \texttt{begingroup}\ \langle\text{statement list}\rangle\ \langle\text{statement}\rangle\ \texttt{endgroup}\end{aligned}$$

That last statement inside the group should be a valid statement on it own, but it can also be empty.

Grouping in MetaPost is a bit unusual (yet another way in which MetaPost is unusual!) in that the `begingroup` ... `endgroup` block is not only usable as a list of ⟨statement⟩s grouped together, it can also be used as an ⟨expression⟩. And when viewed as an expression (which is usually the case for `vardef` macro expansions, but you can also write explicit group blocks in the middle of an equation, or as the body of any type of macro), all the statements in the group are executed as normal, but the last expression inside the group (which could be empty) is taken as the value to use for the expression outside of the group. And precisely that oddity of grouping is what makes `vardef` definitions syntactically equivalent to variables.

Formally, all of the expression syntaxes also have an extra `begingroup` block. For example, ⟨numeric expression⟩ also has:

```
⟨numeric atom⟩ → ⟨numeric token⟩
        | ( ⟨numeric expression⟩ )
        | begingroup ⟨statement list⟩ ⟨numeric expression⟩ endgroup
        | ⟨...⟩
```

and likewise for all other expression types: at the bottom level, there is an `begingroup` ...`endgroup` that is equivalent with a delimited group like `(\<numeric expression>)`. The statements in the ⟨statement list⟩ are executed, but not seen by the expression parser.

The extra grouping usually will not matter, but it means you cannot do things like:

```
vardef stuff =
  fill unitsquare
enddef;
stuff withcolor green;
```

which makes sense once you realize that `vardef` is supposed to equate to a variable. If we assume for a moment that there was instead a normal path variable named `stuff`, the statement would look like this:

```
fill stuff withcolor green;
```

and indeed, after adjusting the `vardef` to:

```
vardef stuff =
  unitsquare % earlier 'fill' deleted
enddef;
```

it works just fine. This is a silly example, of course, but the point to remember is that the last line in a `begingroup` ...`endgroup` should produce a valid expression (which may be empty).

The main point of `begingroup` ...`endgroup` is so that you can save and temporarily redefine variables and internals. But it creates only an implicit grouping; nothing is automatically saved. If you want to save an outside variable or internal, you have to explicitly use `save` or `interim`.

The above rules for the final parts inside of a group block make grouping behave similar to an anonymous function call with one return value; or as a named function, when using `vardef` or the result of a `def` with a `begingroup` ...`endgroup` block around the whole replacement text.

Additionally, because the expression parser does not 'see' the ⟨statement list⟩, you can do complicated things right in the middle of an equation. The plain MetaPost macro named `hide` makes use of that:

```
def hide(text t) = gobble begingroup t; endgroup enddef;
def gobble primary g = enddef;
```

(this is the example definition from the MetaFont book, the actual definition is trickier).

The `begingroup` ...`endgroup` block inside `hide` always results in an empty expression because of the explicit `;` at the end. But an empty expression is still an expression, so that is why the `gobble` macro is needed to 'eat' that empty expression.

One final note about vardef: the addition of begingroup ...endgroup around the ⟨replacement text⟩ is literal. If you wanted to, you could write a vardef including endgroup ...begingroup to temporarily escape to the outer group. But of course then the expansion would not be a valid ⟨... expression⟩ any more, so you could not use that macro in the middle of an expression.

Replacement text details

A ⟨replacement text⟩ is stored for later use without any expansion at definition time. In almost all cases, the meaning of the symbolic tokens will be looked up and applied at expansion time. However, some tokens that may occur inside the ⟨replacement text⟩ have to be interpreted by the program right away to avoid internal confusion:

- def, vardef, primarydef, secondarydef and tertiarydef are the start of an embedded definition.
- enddef ends the ⟨replacement text⟩ unless it matches an embedded definition that started in the previous rule.
- Each ⟨symbolic token⟩ that stands for a macro parameter is changed into a placeholder for that parameter, for later substitution at replacement time.
- quote prevents any of the previous rules applying to the next token. After afterward, the quote token itself is removed from the replacement.

In all the above cases, the check is made for the meaning of the token, not its literal representation. In other words, prior use of let can alter the list of 'keywords'.

The preceding rules mean that

```
def bfour =
  def b = 4 enddef
enddef;
```

is allowed. Any subsequent use of bfour in the program expands into a definition that makes b be a macro that expands to the value 4.

But:

```
def defbfour =
  def b = 4
enddef;
```

will fail, because the definition of defbfour does not end. The enddef stops the embedded definition of b.

To get around this, you can write:

```
def defbfour =
  quote def b = 4
enddef;
```

which is a valid definition of defbfour. Now you will have to use defbfour enddef; when using defbfour later, of course. Otherwise the embedded definition of b never ends.

The existence of quote allows some special syntaxes. With the above definition, you could specify

```
defbfour *4 enddef;
```

which would define b to be a macro with replacement text 4*4. Admittedly, this is not very useful but I want to document everything related to definitions, and the use of quote cannot be omitted.

Formal definition syntax

```
⟨macro definition⟩ → ⟨macro heading⟩ = ⟨replacement text⟩ enddef
⟨macro heading⟩ → def ⟨symbolic token⟩ ⟨delimited part⟩ ⟨undelimited part⟩
      | vardef ⟨declared variable⟩ ⟨delimited part⟩ ⟨undelimited part⟩
      | vardef ⟨declared variable⟩ @# ⟨delimited part⟩ ⟨undelimited part⟩
      | ⟨binary def⟩ ⟨parameter⟩ ⟨symbolic token⟩ ⟨parameter⟩
⟨delimited part⟩ → ⟨empty⟩
      | ⟨delimited part⟩ (⟨parameter type⟩ ⟨parameter tokens⟩)
⟨parameter type⟩ → expr | suffix | text
⟨parameter tokens⟩ → ⟨parameter⟩ | ⟨parameter tokens⟩, ⟨parameter⟩
⟨parameter⟩ → ⟨symbolic token⟩
⟨undelimited part⟩ → ⟨empty⟩
      | ⟨parameter type⟩ ⟨parameter⟩
      | ⟨precedence level⟩ ⟨parameter⟩
      | expr ⟨parameter⟩ of ⟨parameter⟩
⟨precedence level⟩ → primary | secondary | tertiary
⟨binary def⟩ → primarydef | secondarydef | tertiarydef
```

Final words

You now know all about how to define your own MetaPost macros, in theory. But the best way to learn is by doing and making mistakes, and that is definitely the case here as well. When I started using MetaPost in earnest, at first nothing I tried seemed to work. Remembering my own initial frustrations about anything more than trivial use of the MetaPost programming language is what prompted me to write these papers. I hope they will be helpful to you.

Taco Hoekwater

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.

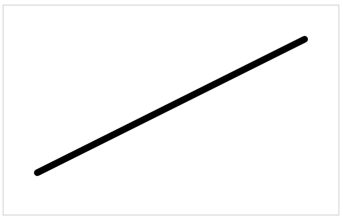
To get it out of the way, here is the syntax definition for path expressions:

```
⟨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.

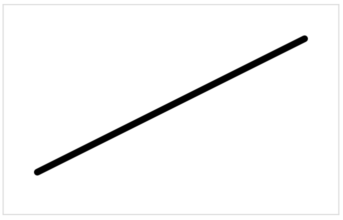
The simplest case is just from a few pair primaries and a basic path join:

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



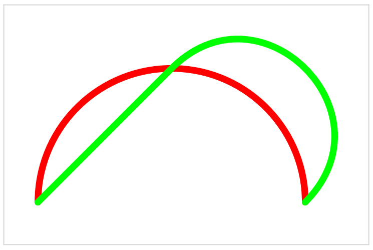
Nearly as easy is to form a path from a path (sub)expression:

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



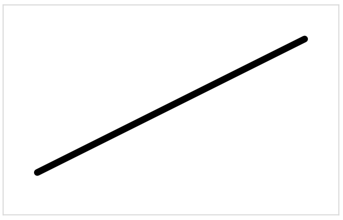
This part of the syntax is about adding `()` grouping, just like in calculus. It makes sure that the expression inside the parentheses is converted into a path first, before any of the outer processing happens. This can be useful because in some cases subsequent `path join`s can have an effect on prior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



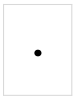
Of course, pairs can also be specified using variable names (we will look at the formal syntax of pairs later):

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



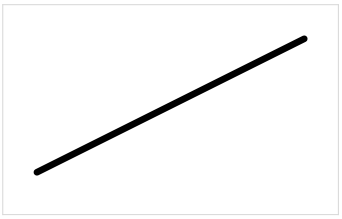
Or even from a single point:

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



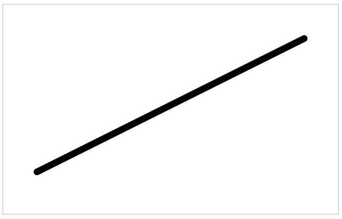
Or from another path:

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



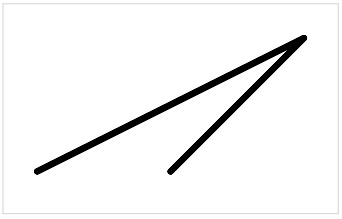
Or from the reverse of another path:

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

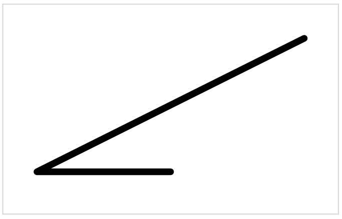


While there is no visual difference in the above image, as a result of the reverse operator the beginning and end of the path q have been flipped, as can be clearly seen from the following examples:

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

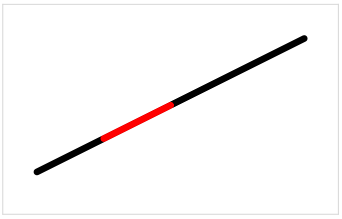


```
path p, q;
q = (0,0)..(200,100);
p = reverse q--(100,0);
draw p;
```



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

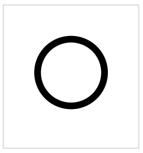
```
path p, q;
q = (0,0)..(200,100);
p = subpath (0.25,0.5) of q;
draw q;
draw p withcolor red;
```



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.

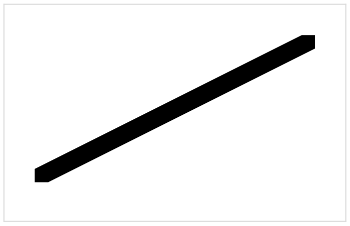
And from a pen:

```
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;
```

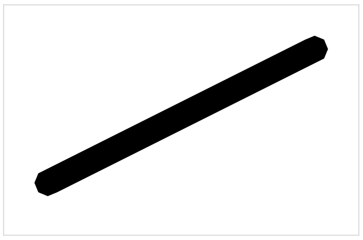


This last option for path construction comes with a few warnings and hints:

□ 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.

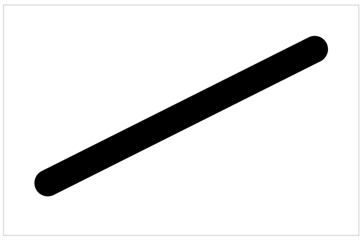
  Here are two workarounds, both of which make use of primitives that have not been covered yet. However, the function of these primitives should hopefully be clear from the examples.
  First, if actual precision in the curves is not very important, you can get away with converting the pencircle to a path, 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;
```



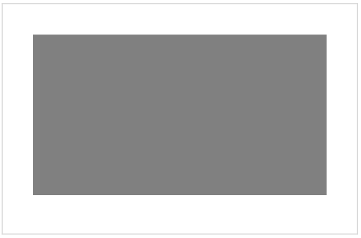
  Or second, if greater curve precision is indeed needed, you can create a pen with a larger number of vertices by using a for loop to construct a new path from the pencircle:

```
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;
```

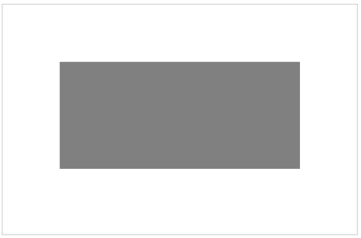


☐ To get the ‘other’ side of a cyclic path, reverse the path:

```
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;
```

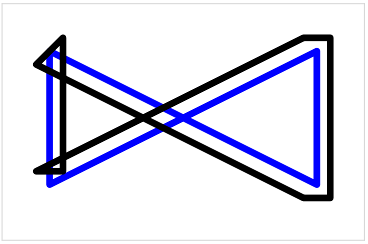


☐ The path created by envelope is not always completely ‘clean’ or even ‘correct’ in the current version of MetaPost; it is intended to produce the visual envelope, not the ‘best’ path to create that envelope. It also still has bugs, especially with self-intersecting paths.

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.

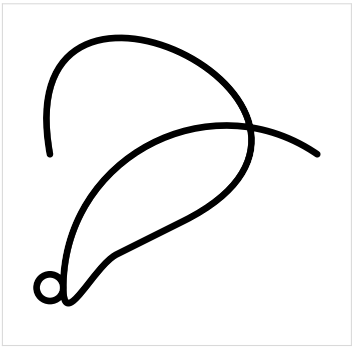
You can see some of the problems that may occur in the next example:

```
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

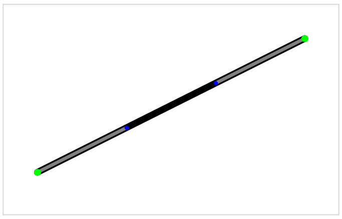
This section is about how to control the shape of a path: how the points are connected to form actual curves. Each point will be connected to the next point by a Bézier curve segment that is controlled by the two points themselves, and two extra 'control points'.

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

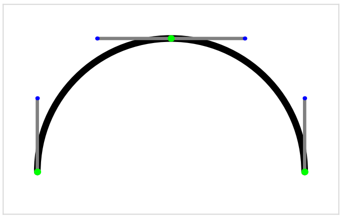
We will start by talking about direction specifiers. In the simplest case, the example below shows an empty direction specifier. Here two pairs are joined just by the two dots that we frequently saw in the previous section.

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



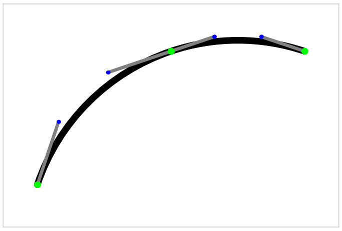
In this example, we are letting MetaPost decide by itself how it wants to connect the pairs. If there are only two pairs, the result is a straight line connecting them. If there are more than two more pairs in the sequence that are not in a straight line, MetaPost will try to create a nice curve connecting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

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



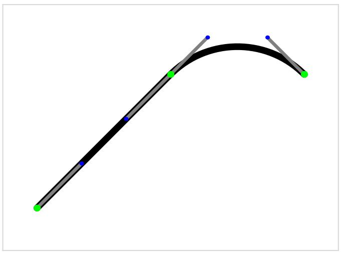
When left to its own devices, MetaPost will try to connect the points in such a way that the overall direction of the combined curve along the path only changes in a fluid way. The curvature at the start- and endpoints will attempt to follow a circular arc through the initial set of pairs. This can affect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

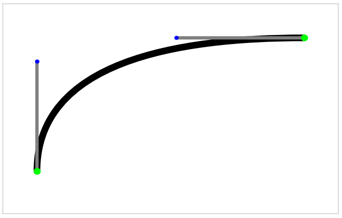
```
path p;
p = ((0,0)..(100,100))..(200,100);
detaileddraw p;
```



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.

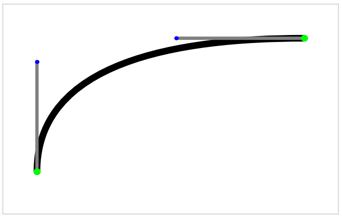
Direction vectors If you want to have explicit direction control, you can add a direction specifier, for example, using pair expressions that represent direction vectors:

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



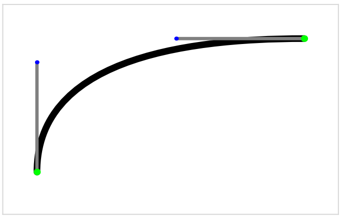
The previous example is not very readable; it is easier to understand using pair expressions stored in variables:

```
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;
```

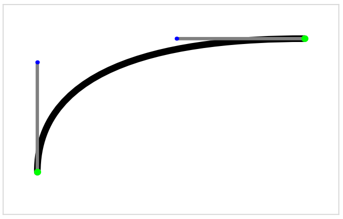


It should be obvious that the vector has to be completely known. If not, an error will be generated. On the other hand, a vector of `(0,0)` is acceptable. This has the same effect as not specifying a vector at all.

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;
```



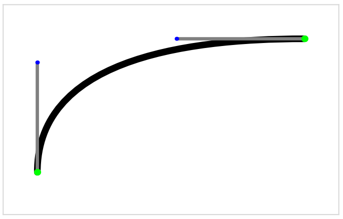
The actual definition of `dir` could be:

```
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;
```

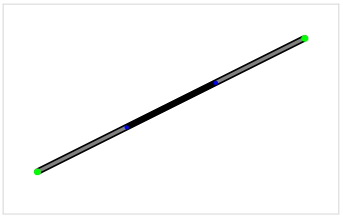


When building paths inside macros, either one or the other of these cases (`pair` or `numeric`) may be easier to work with. The end results are identical; there is always an internal vector constructed.

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.

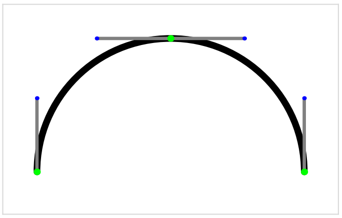
- It sets the amount of curliness (angle) at that point.
- If the requested amount of curl is high, it will adjust the curliness at adjacent points as well.
- Its assumed default value at ending points is 1.
- An explicit `curl` setting makes that point an ‘endpoint’ (a.k.a. a corner).

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.

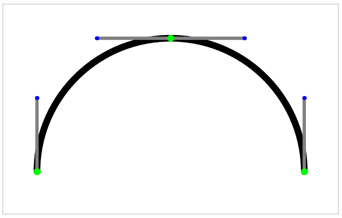
So:

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



has the same effect as:

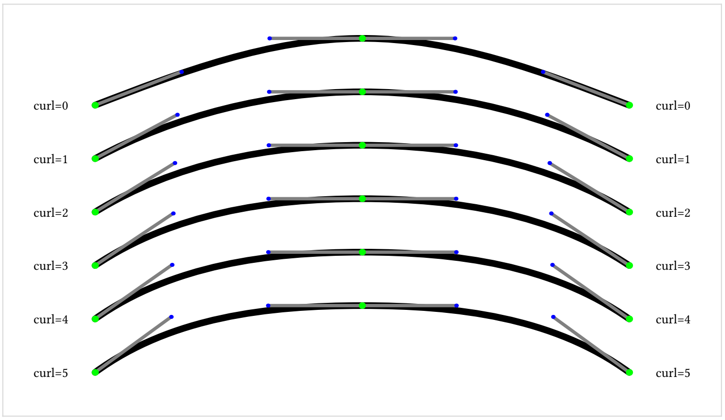
```
path p;
p = (0,0){curl 1}..
    (100,100)..
    {curl 1}(200,00);
detaileddraw p;
```



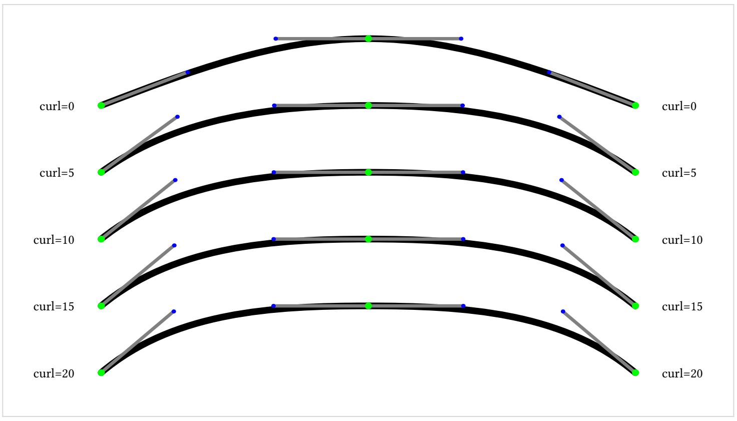
For a better understanding, it is easier to start by showing a progression. The next output was created using this example:

```
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
```

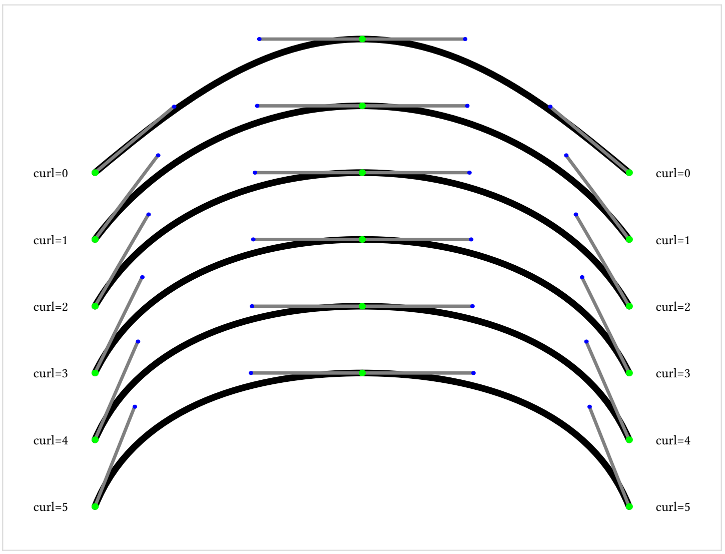
There is some extra stuff there to print the labels, but the main thing the example does is create a set of paths with increasing settings for `curl` at both of the endpoints:



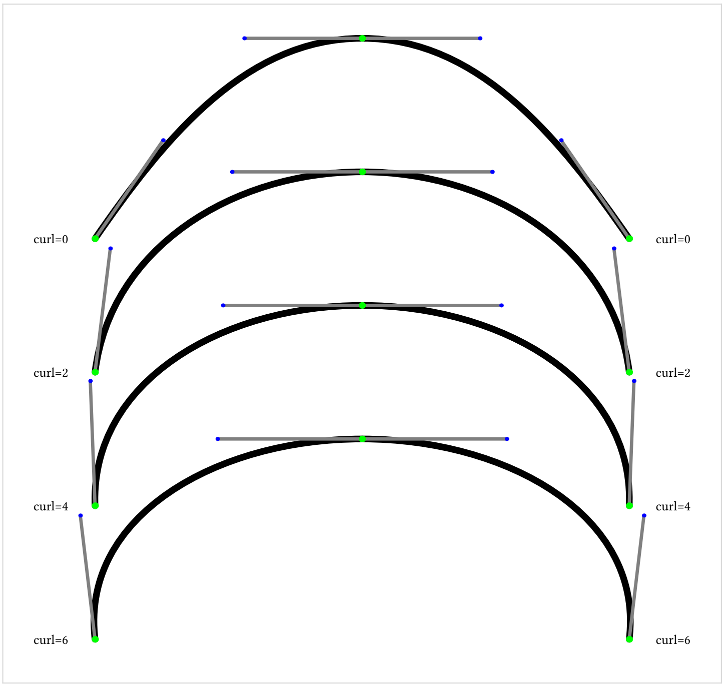
As you see, the angle (to the control point) increases when the `curl` value rises. But there is an upper limit above which it does not matter any more how much higher the `curl` value gets. Above a certain value, there is not enough curvature left for `curl` to be able to redistribute towards the endpoints:



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:

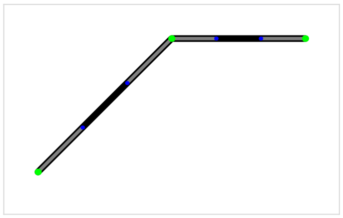


While the curl specifier takes some getting used to, it is a good tool to control the initial and final curves of a path since using a direction vector is not always straightforward, especially so if paths are rotated or otherwise transformed. On the downside, it may seem a bit temperamental, because 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);
```

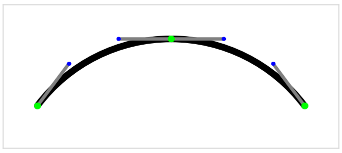
which makes every point a corner. The segments between those points is then filled using the '2 point' segment logic, resulting in straight lines.

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.

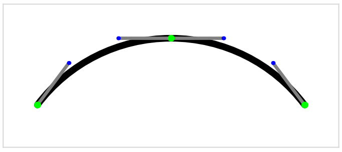
In an example:

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



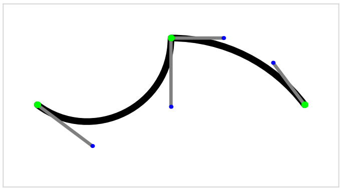
is the same as:

```
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;
```

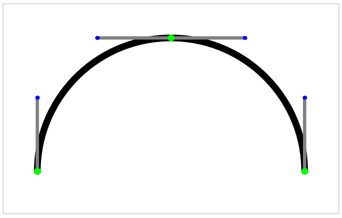


Path connectors

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:

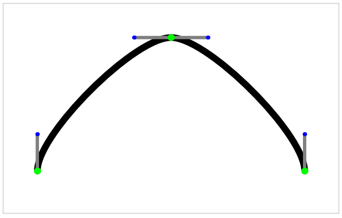
```
path p;
p = (0,0)..(100,100)..(200,0);
detaileddraw p;
```



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

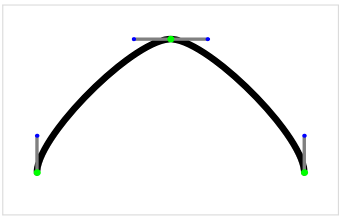
Amongst other things, the tension settings control how ‘tight’ the path is between two points:

```
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;
```



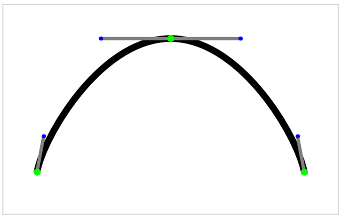
If you say just `tension 2`, it is silently interpreted as `tension 2 and 2`.

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).

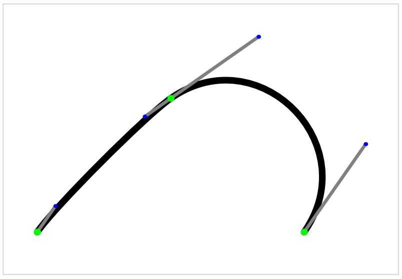
An example of that potential effect on the direction can be seen in the following code, where the two tension values for each segment are not identical:

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



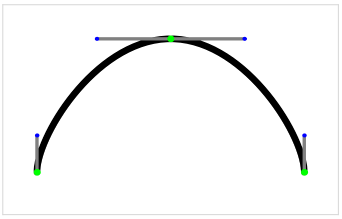
Another example is when all the segments do not have the same/complementary tension settings:

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

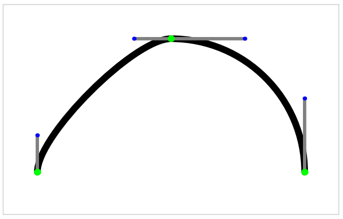


If we 'fixate' those examples by filling in explicit direction vectors:

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



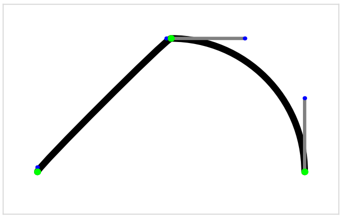
```
path p;
p = (0,0){up}..tension 2 ..
    (100,100){right}..
    {down}(200,0);
detaileddraw p;
```



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:

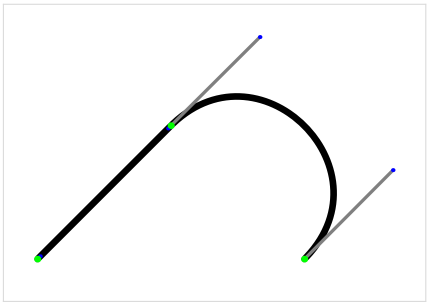
```
path p;
p = (0,0){up}..tension 16 ..
    (100,100){right}..
    {down}(200,0);
detaileddraw p;
```



the vector is now 1⁄16 of its ‘normal’ length, and it becomes almost a straight line.

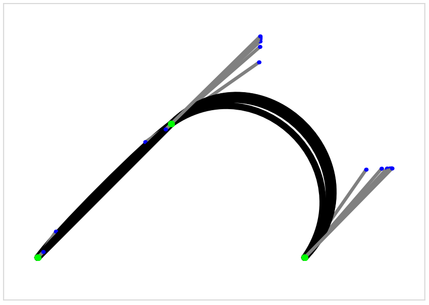
If there are multiple non-fixated directions, the vector changes become more complicated because in that case, the extra effect of the tension settings on the first segment is that the control vectors for the other segments become a bit longer:

```
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:

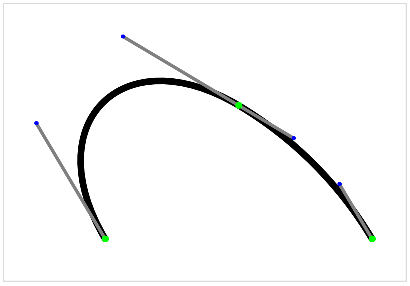
```
path p,q;
for i = 2 upto 8:
  p := (0,0)..tension i ..
    (100,100)..
    (200,0);
  detaileddraw p;
endfor
```



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;
```

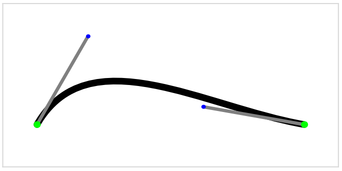


But lowering the tensions (and thus increasing the length of the vectors) also increases the chances that the internal calculations that control the behaviour of the curve become unpredictable. For that reason, the lowest value you are allowed to set `tension` to is `0.75`.

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.

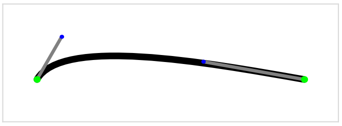
Sometimes the latter results in sub-optimal curves, like in the following example where we may not want an inflection to happen:

```
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.

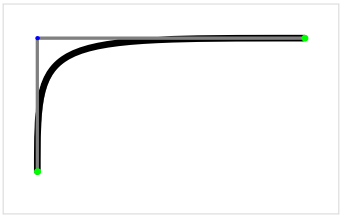
To wrap up the discussion of `tension`: here are two macros that are usually predefined:

```
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.

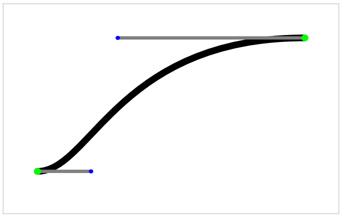
For such cases, MetaPost allows you to input explicit control point values, either by using a single point, like:

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



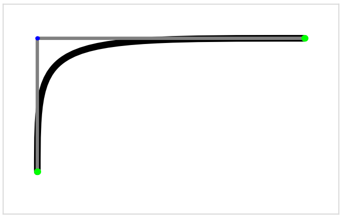
or by using two separate points:

```
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;
```

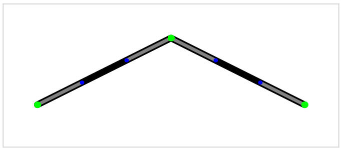


Handy to know:

- □ 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.

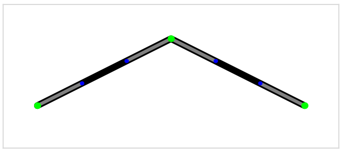
Path concatenation Just like with strings, it is possible to concatenate two paths by using the `&` operator:

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



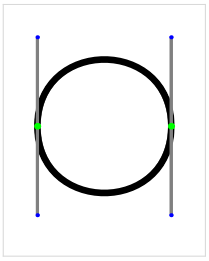
This only works if the left and right points are identical, and it is equivalent to

```
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;
```

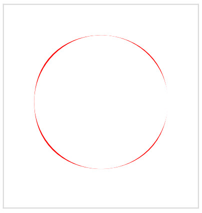


The `cycle` operator adds a reference back to the first point of the path being created, but it also adds a special marker to the internal path structure. Without it, the path is not considered to be cyclic, and you cannot use it with e.g. `fill`.

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:

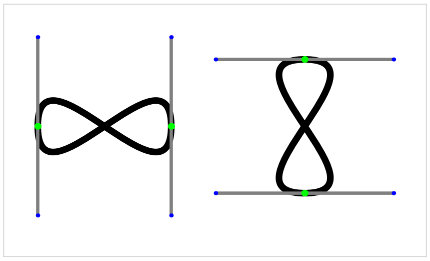
```
path p;
p = (0,0)..{down}(100,0) .. cycle;
fill p withcolor red;
draw origin shifted (50,0)
  withpen pencircle scaled 100
  withcolor white;
```



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

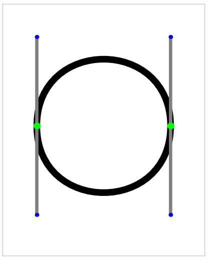
You may be wondering about the use of `eps` (defined as `.00049`) one example back. The reason is that while a two-point path can (and usually does) define a path that turns 360 degrees (by moving upward through the first point and then downward through the second point), it can also define a path that 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`.

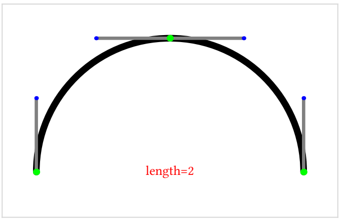
So now you have all the tools to define a path.

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:

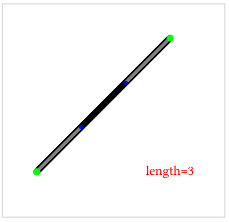
- the `length` of a path is the number of segments it consists of, which is equal to the number of explicit points for cyclic paths, and for non-cyclic paths, the same number minus one.

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

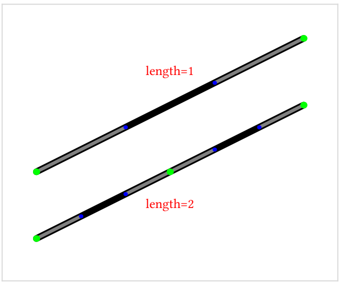
  I brought this up now instead of in the section on operations because it is important to know that 'empty' curve segments *do* count when you define a path, so:

  ```
  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, `pair`s 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.

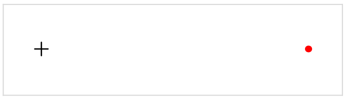
For reference, let's start with the formal definition:

```
⟨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;
```

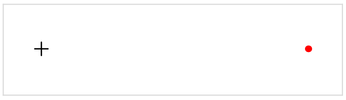


The examples below will always print the 'active' pair in red, and the 'origin' point at `(0,0)` in black; and occasionally an extra dot or path is drawn as well. The red dot is always the target of the example.

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.

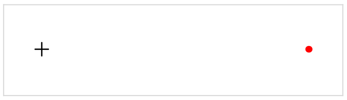
You can also define a pair from another pair variable:

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



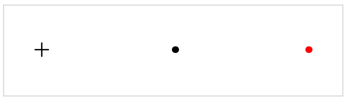
In fact, because of the way collections of equations are solved in MetaPost, you can invert the equation that gives a value to `b` and the equation that equates `a` to `b`. As long as the equation is resolved before you try to draw the dot, the order of the equations is irrelevant. This is generally 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;
```



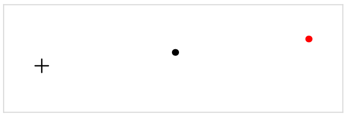
You can define a pair from an expression by adding parentheses:

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



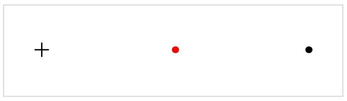
The internal expression can then do all of the things from the full syntax. Besides addition, for example, you could also do multiplication:

```
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:

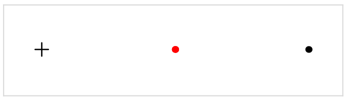
```
pair a;
a = .5[(0,0),(200,0)];
draw_origin;
drawdot (200,0);
drawdot a withcolor red;
```



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).

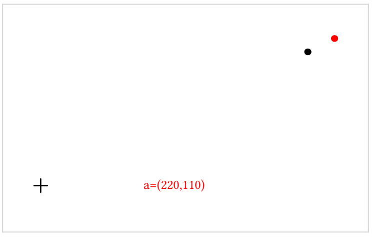
Here some equivalents of the previous example:

```
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;
```



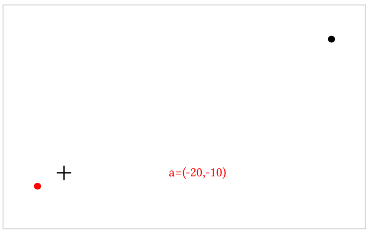
It is important to realize that while we informally call this the ‘off-the-way’ operator, it is really just a multiplier along a line where the first listed pair is at ‘distance’ zero and the second listed pair is at 'distance' one. If the points are named a and b and the multiplier is x, then it calculates $a + x * (b - a)$. 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:

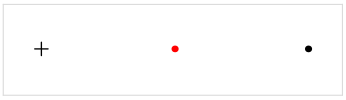
```
pair a;
a = -0.1[(0,0),(200,100)];
draw_origin;
drawdot (200,100);
drawdot a withcolor red;
labelat((100,0), "a=(-20,-10)")
```



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.

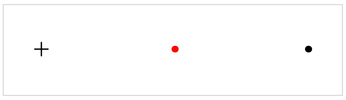
We can also define a new pair from a multiplication applied to another pair:

```
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;
```



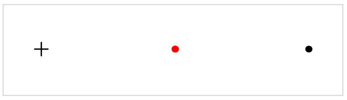
The formal syntax rules here are a little contrived, but the end result is that this operation is quite like the 'off-the-way' operator, except that variables are not allowed.

$\langle$scalar multiplication operator$\rangle$ $\longrightarrow$ $\langle$plus or minus$\rangle$
   | $\langle$numeric token primary not followed by + or - or a numeric token$\rangle$

$\langle$numeric token primary$\rangle$ $\longrightarrow$ $\langle$numeric token$\rangle$/$\langle$numeric token$\rangle$
   | $\langle$numeric token not followed by '/ numeric token' $\rangle$

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:

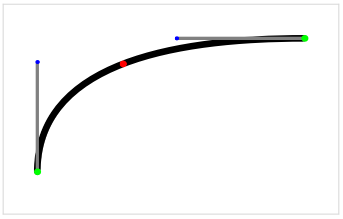
```
>> (20,20)
>> (19.9997,19.9997)
```

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.

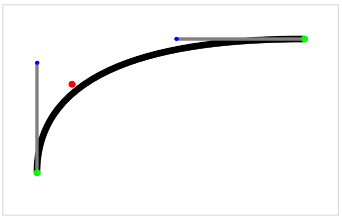
The next two examples deal with defining a new pair based on some part of a path. First, you take any specific point along a path (it does not have to be an integer value):

```
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:

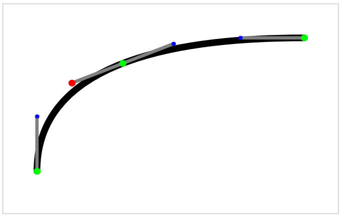
```
path p;
pair a;
p = (0,0){up}..{right}(200,100);
a = precontrol 0.5 of p;
detaileddraw p;
drawdot a withcolor red;
```



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:

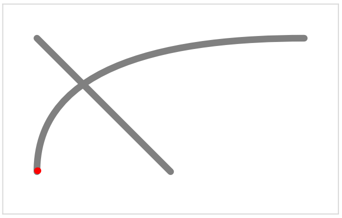
```
path p,q;
pair a;
p = (0,0){up}..{right}(200,100);
q = subpath (0,0.5) of p
    & subpath(0.5,1) of p;
a = precontrol 1 of q;
detaileddraw q;
drawdot a withcolor red;
```



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;
```

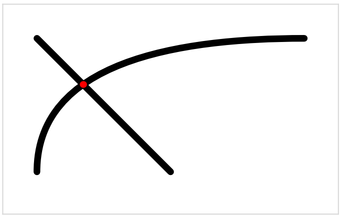


No, that is not a typographical error in the example!

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.

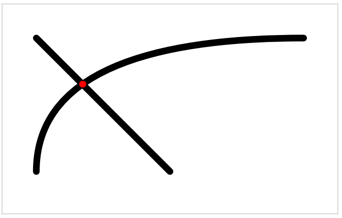
If you want to have an actual point, you have to fetch it from the path using the point operator:

```
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;
```



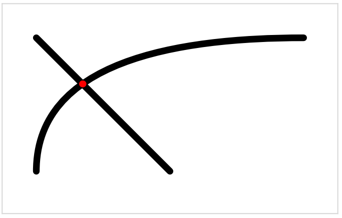
But you need to be careful using it because, owing to the definition of intersectionpoint, you will usually get a point that is not on *either* path. The typical definition of this macro tries to give you a point that is very close to both of the two points. The simplified definition looks like this:

```
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:

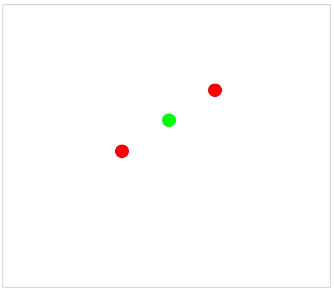
```
path p, q;
pair a, b, c;
p = (0,0){up}..{right}(200,100);
q = (0,100)..(50,50)..(100,0);
a = p intersectiontimes q;
b = point (xpart a) of p;
c = point (ypart a) of q;
draw p; draw q;
drawdot b withcolor red;
drawdot c withcolor red;
```



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;
```

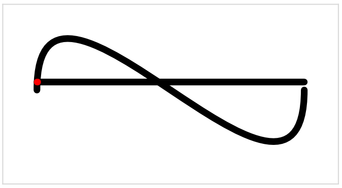


If there are no intersections, `intersectiontimes` returns `(-1,-1)`.

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:

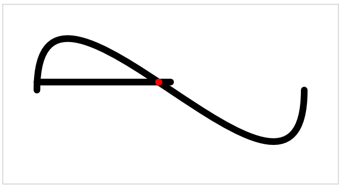
```
pair a;
path p, q;
p := (0,0){up}..{up}(200,0);
q := (200,6)..(0,6);
a := p intersectiontimes q;
draw p; draw q;
drawdot (point (xpart a) of p)
        withcolor red;
```



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.

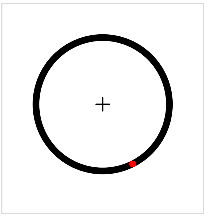
The intersection times along `p` are the same in both cases: `0.01573` and `0.46989`.

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':

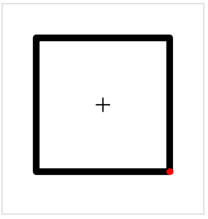
```
path p;
pair a;
a = penoffset (1,0.5) of
      pencircle scaled 100;
p = makepath pencircle scaled 100;
draw p;
draw_origin;
drawdot a withcolor red;
```



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.

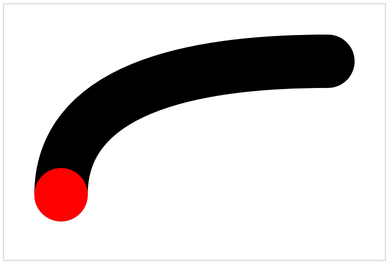
Pens are the objects that are used to 'trace' paths when you use `draw` or `filldraw` (or rather the underlying primitive `addto`). MetaPost has two different kind of pens: pens that derived from a circle (a.k.a. elliptical pens) and pens that are based on a convex cycle of straight segments (a.k.a.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

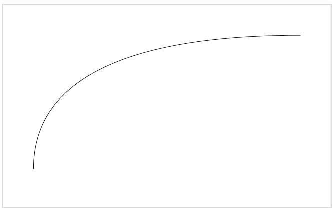
```
pen mypen;
mypen = pencircle scaled 40;
draw (0,0){up}..{right}(200,100)
  withpen mypen;
fill makepath mypen withcolor red;
```



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:

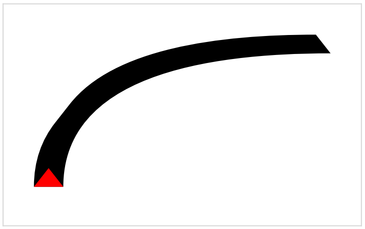
```
pen mypen;
mypen = nullpen scaled 4000;
draw (0,0){up}..{right}(200,100)
  withpen mypen;
fill makepath mypen withcolor red;
```



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;
```



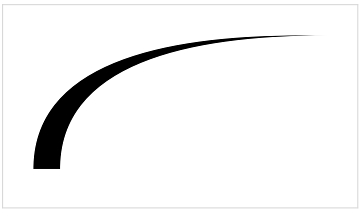
The result of makepen is always a polygonal pen. It is not possible to construct elliptical pens in this way as they *have* to be based on pencircle.

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:

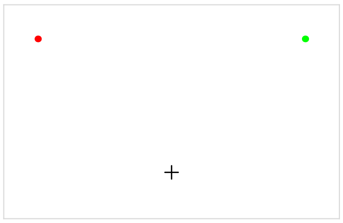
```
⟨transform primary⟩ → ⟨transform variable⟩ | ⟨transform argument⟩
    | ( ⟨transform expression⟩ )
    | begingroup ⟨statement list⟩⟨transform expression⟩ endgroup

⟨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`).

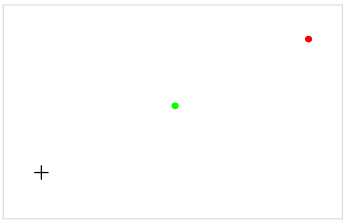
rotated works counter-clockwise around the origin:

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



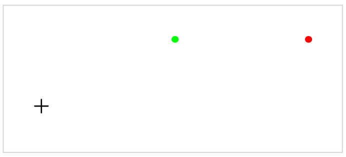
scaled multiplies the separate components:

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



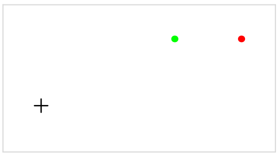
shifted moves things around:

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



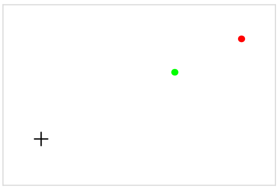
slanted slants things by adding some multiple of the y value to the x value:

```
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.

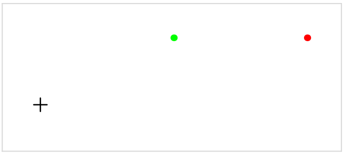
The identity transformation could also have been defined like this:

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

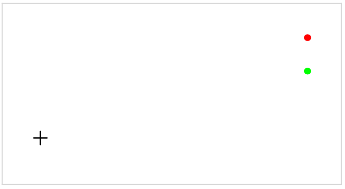
but those six equations are not nearly as cute or fun to explain.

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

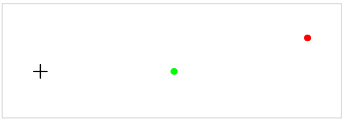
```
pair a;
a = (100,50) xscaled 2;
draw_origin;
drawdot (100,50) withcolor green;
drawdot a withcolor red;
```



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



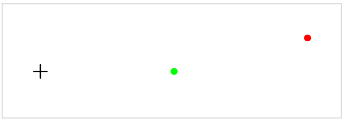
```
pair a;
a = (100,0) zscaled (2,0.25);
draw_origin;
drawdot (100,0) withcolor green;
drawdot a withcolor red;
```



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:

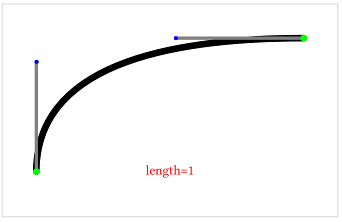
- ☐ You can chain transformers, they are processed left to right.
- ☐ There is no direct assignment syntax for `transform` type definitions: you have to modify an existing transform, build one using explicit ⟨transformer⟩ equations, or assign each of the six parts using separate equations.
- ☐ Don't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

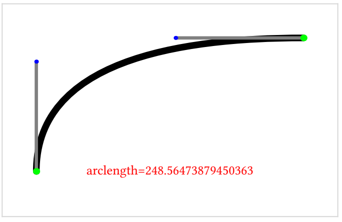
```
path p; numeric d;
p = (0,0){up}..{right}(200,100);
d = length p;
detaileddraw p;
labelat((100,0), "length="&decimal d);
```



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:

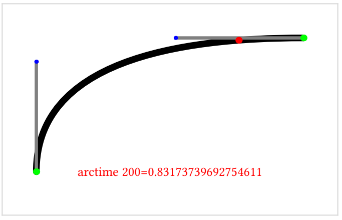
```
path p; numeric d;
p = (0,0){up}..{right}(200,100);
d = arclength p;
detaileddraw p;
labelat((100,0), "arclength="&decimal d);
```



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

Find a specific drawn time of a path:

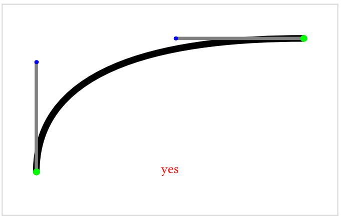
```
path p; numeric d;
p = (0,0){up}..{right}(200,100);
d = arctime 200 of p;
detaileddraw p;
labelat((100,0),
        "arctime 200="&decimal d);
drawdot (point d of p) withcolor red;
```



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

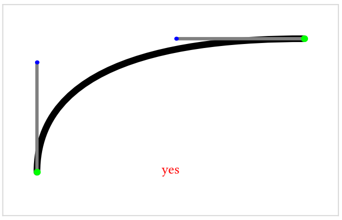
Test if a variable is a path:

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



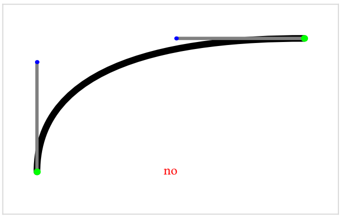
The if command tests the type of the following expression. This means that single pairs fail even though they are valid as path declarations (due to the automatic conversion into a path when an assignment takes place). But on the other hand, it means that you can use an explicit expression:

```
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:

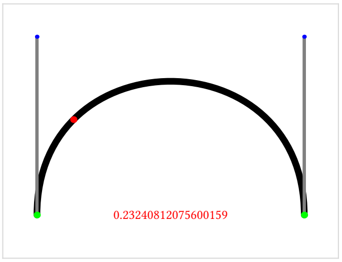
```
path p;
p = (0,0){up}..{right}(200,100);
detaileddraw p;
if cycle p:
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```



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;
```



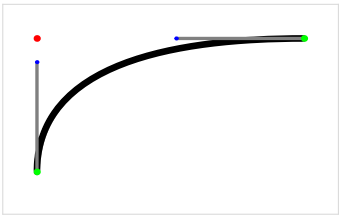
Side note: in this example it is obvious that a Bézier curve is *not* the same as a circular arc. If they were, the return value would have been exactly `0.25`.

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;
```



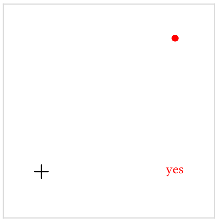
Also defined are the complementing primitives `llcorner`, `lrcorner`, and `urcorner`.

Pair operations

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

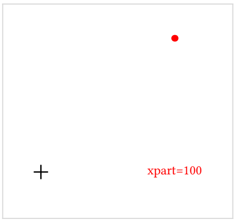
Test if a variable is a pair:

```
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:

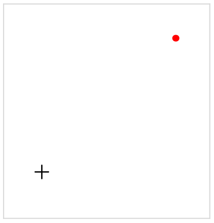
```
pair a; numeric d;
a = (100,100);
d = xpart a;
draw_origin;
drawdot a withcolor red;
labelat((100,0), "xpart=" & decimal d);
```



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

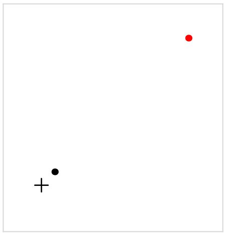
You can multiply or divide a pair by a numeric:

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



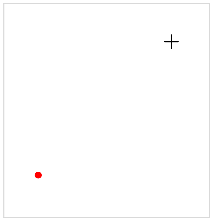
You can add or subtract another pair:

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



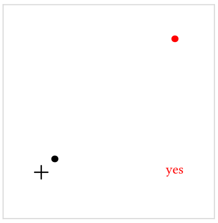
You can negate a pair:

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



You can compare a pair to another pair:

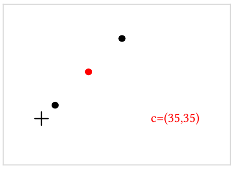
```
pair a,b;
b = (10,10);
a = (100,100);
draw_origin;
drawdot b;
drawdot a withcolor red;
if a > b:
  labelat((100,0), "yes");
else:
  labelat((100,0), "no");
fi
```



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

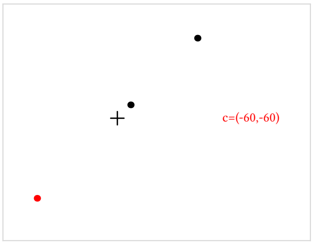
You can mediate between two pairs using the off-the-way operation:

```
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)")
```

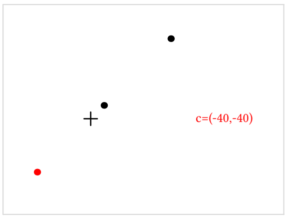


The result here is `(-60,-60)` because the mediation is processed first (using the positive value of `1`):

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

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)")
```

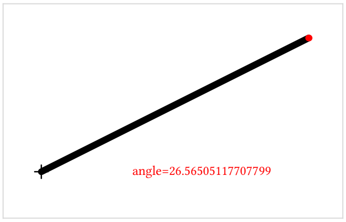


The result is (-40,-40) because the mediation is processed with the value -1:

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

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)
```

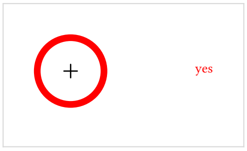


Pen operations

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.

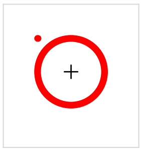
Test if a variable is a pen:

```
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

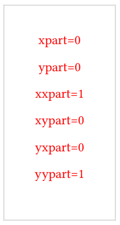
Now let's look at the operations you can do on `transform`s. Like with pens, there are not a whole lot of them.

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 `path`s, `pair`s, `pen`s, and `transform`s.

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

Conditions and loops

Abstract

This article is about how to make your program decide what to do next: conditions and loops.

Conditions

Conditions in MetaPost are both simple and a bit unexpected.

They are simple because there is only one command: `if`. The syntactical structure of that command is simple as well: it is the keyword `if` followed by a condition test that is closed off by a colon, then the replacement body and finally it ends with the closing keyword `fi`. And as one would expect, conditions can contain nested conditions, and there are provisions for alternatives (`else` and `elseif`).

The unexpected bit: conditions can be inserted (almost) everywhere and do not have to adhere to syntactical structure rules except for their own internal ones. For example, a nested condition can start halfway through the condition test and end somewhere in the middle of the replacement text of the outer condition. This allows for a very flexible but also sometimes a little confusing or potentially obscure input code. I find it helps to think of each `if` as an in-line preprocessor that stops at the next `fi`.

First, here is the formal definition of ⟨condition⟩:

```
⟨condition⟩ → if ⟨boolean expression⟩ : ⟨conditional text⟩⟨alternatives⟩ fi

⟨alternatives⟩ → ⟨empty⟩
  | else : ⟨conditional text⟩
  | elseif ⟨boolean expression⟩ : ⟨conditional text⟩⟨alternatives⟩

⟨boolean primary⟩ → ⟨boolean variable⟩
  | true
  | false
  | ( ⟨boolean expression⟩ )
  | begingroup ⟨statement list⟩⟨boolean expression⟩ endgroup
  | known ⟨primary⟩
  | unknown ⟨primary⟩
  | ⟨type⟩⟨primary⟩
  | cycle ⟨primary⟩
  | odd ⟨numeric primary⟩
  | not ⟨boolean primary⟩
  | bounded ⟨primary expression⟩
  | clipped ⟨primary expression⟩
  | filled ⟨primary expression⟩
  | stroked ⟨primary expression⟩
  | textual ⟨primary expression⟩

⟨boolean secondary⟩ → ⟨boolean primary⟩
  | ⟨boolean secondary⟩ and ⟨boolean primary⟩

⟨boolean tertiary⟩ → ⟨boolean secondary⟩
  | ⟨boolean tertiary⟩ or ⟨boolean secondary⟩
```

$\langle$boolean expression$\rangle \rightarrow \langle$boolean tertiary$\rangle$
  | $\langle$numeric expression$\rangle\langle$relation$\rangle\langle$numeric tertiary$\rangle$
  | $\langle$pair expression$\rangle\langle$relation$\rangle\langle$pair tertiary$\rangle$
  | $\langle$transform expression$\rangle\langle$relation$\rangle\langle$transform tertiary$\rangle$
  | $\langle$boolean expression$\rangle\langle$relation$\rangle\langle$boolean tertiary$\rangle$
  | $\langle$string expression$\rangle\langle$relation$\rangle\langle$string tertiary$\rangle$

$\langle$relation$\rangle \rightarrow$ < | <= | > | >= | = | <>

Condition tests

As you can see above, the ⟨boolean variable⟩s `true` and `false` are primitive keywords:

```
if true:
  message "hi";
fi
```

Of course, this is a silly example.

However, new boolean variables can be declared:

```
boolean mystate;
mystate = true;
```

Boolean variables can then be used in `if` expressions:

```
if mystate:
  message "hi";
fi
```

Note that declared boolean variables start off in the `unknown` state, just like all other declared variables.

If you really want to, you can use parentheses to create a nested ⟨boolean expression⟩:

```
if (mystate):
  message "hi";
fi
```

But as mentioned in the first paragraph of this article, `if` can be nested inside another `if` without needing extra parentheses, so

```
if (if mystate: false else: true fi):
  message "hi";
fi
```

and

```
if if mystate: false else: true fi:
  message "hi";
fi
```

are equivalent. Usually MetaPost programmers do not use parentheses in situations like this, because parentheses can be easily misunderstood as the syntax for a `pair`. But in some cases, parenthesis might be needed to resolve syntactic precedence.

This was another silly example: the `if mystate: false else: true fi` condition can be written much clearer and shorter using `not mystate` instead (see below).

More important is that you can use grouping, because that allows execution of extra statements ‘on the fly’:

```
if begingroup
     mystate := false ;
     mystate
   endgroup:
   message "hi";
fi
```

You can test whether a conditional value is (un)known:

```
if known mystate:
  message "known";
fi
if unknown mystate:
  message "unknown";
fi
```

Boolean variables are `unknown` unless initialized, but indeed `known` when they are `false` as well as when they are `true`.

You can test for the variable type:

```
if boolean mystate:
  message "boolean";
fi
```

this works for all other variable types as well (`if path mystate:` et cetera).

You can ask if something is a cyclic path:

```
if cycle fullcircle:
  message "cyclic path";
fi
```

For ease of use this test works on anything, but of course it is only true for cyclic paths.

You can ask if a ⟨numeric primary⟩ is odd:

```
if odd 5.5:
  message "odd";
fi
```

A non-integer numeric is rounded before testing for even or oddness. However, the rounding rule in MetaPost is a little weird: for halfway cases like this one, the `odd` test rounds rigorously upward to the nearest integer before it decides, so while 5.5 is even, -5.5 is odd.

That is just for the halfway cases, though:

```
if odd -5.5004:
  message "odd";
else:
  message "even";
fi
```

will print out the string even, because -5.5004 rounds to -6 as one would expect.

A ⟨boolean primary⟩ can be inverted (as seen earlier):

```
if not known mystate:
  message "known";
fi
```

There are special `if` tests for objects inside pictures (pictures are explained in detail in a different article):

```
if filled p:
   message "filled";
fi
```

There are different keywords for each of the five different types of graphical objects that can be contained inside pictures:

`filled`	true for filled paths
`stroked`	true for stroked paths
`clipped`	true for clip objects
`bounded`	true for setbounds objects
`textual`	true for typeset text

The `textual` test may have unexpected results when you use external processing for included text (for example `btex ... etex` in plain MetaPost or `textext()` in ConTEXt) because such subsystems do not always translate the text to primitive operations in a simple way. The `textual` test works on graphical objects created using the low-level `infont` operation, which may or may not be used by such subsystems.

Actually you can apply these tests not just within `within` (see below about for-loops), but also on an actual complete picture. Here is a simple example:

```
draw fullcircle;
fill fullsquare;
for a within currentpicture:
  if stroked a:
    message "stroked";
  fi
endfor
if stroked currentpicture:
  message "still stroked";
fi
```

This works because if their argument is of type `picture`, the tests test the first item inside that picture.

Going down the syntax tree, a ⟨boolean primary⟩ can be composed of ⟨boolean secondary⟩ using `and`:

```
boolean mycondition;
if mystate and unknown mycondition:
  message "state true but condition unknown"
fi
```

Similarly, a ⟨boolean secondary⟩ can be composed of ⟨boolean tertiary⟩ using `or`:

```
if mystate or unknown mycondition:
  message "state true or condition unknown"
fi
```

And tertiaries can by built up from expressions:

```
if 5 < 6:
  message "universe still sane";
fi
```

relation tests are: < (less than), <= (less or equal), > (greater than), >= (greater or equal), = (equal), and <> (not equal).

Alternatives

There is also a possible `else` clause:

```
if 5 < 6:
  message "universe still sane";
else:
  message "the sky is falling";
fi
```

And lastly, there is a chained `elseif` possible:

```
if 5 < 6:
  message "universe still sane";
elseif mystate:
  message "in limbo";
else:
  message "the sky is falling";
fi
```

where the `elseif`s can be repeated.

There is always a colon required (marking the end of the condition), even in the lone `else` case!

loops

Loops allow bits of code to be repeated until a certain condition is met.

Loops start with a ⟨loop header⟩ (`for...`, see below) and end with `endfor`.

Similar to conditions, loops can be inserted in the input nearly everywhere assuming their replacement text is syntactically valid at that spot, including containing a loop inside of another loop. However, there is a restriction related to conditions: loops cannot be interwoven with the actual syntax of a conditional.

For example, this input generates an error:

```
if true:
  for a = 1, 2: % WRONG!
    elseif a>0:
      message "found";
  endfor
  fi
endfor
```

The error happens because in the first loop iteration, when the alternative text following the `elseif` is processed, MetaPost cannot find its ending command (the `elseif` will not be ‘seen’ until the next iteration, and outer `fi` is unreachable because it is not part of the loop text. You can still think of loops as in-line preprocessors, just be careful if conditionals are also involved.

Here is the formal syntax definitions for ⟨loop⟩:

```
⟨loop⟩ → ⟨loop header⟩ : ⟨loop text⟩ endfor
⟨loop header⟩ → for⟨symbolic token⟩⟨is⟩⟨for list⟩
   | for ⟨symbolic token⟩⟨is⟩⟨progression⟩
   | forsuffixes ⟨symbolic token⟩⟨is⟩⟨suffix list⟩
   | forever
   | for ⟨symbolic token⟩ within ⟨picture expression⟩
⟨is⟩ → = | :=
⟨for list⟩ → ⟨expression⟩ | ⟨empty⟩
   | ⟨for list⟩ , ⟨expression⟩
   | ⟨for list⟩ , ⟨empty⟩
⟨suffix list⟩ → ⟨suffix⟩
   | ⟨suffix list⟩ , ⟨suffix⟩
⟨progression⟩ → ⟨initial value⟩ step ⟨step size⟩ until ⟨limit value⟩
⟨initial value⟩ → ⟨numeric expression⟩
⟨step size⟩ → ⟨numeric expression⟩
⟨limit value⟩ → ⟨numeric expression⟩
⟨exit clause⟩ → exitif ⟨boolean expression⟩ ;
```

Loop commands

Loops can be created using an explicit expression list:

```
for a = "1","2","3":
  message (a);
endfor
```

As shown by the formal syntax, you can use `:=` instead of `=` if you want:

```
for a := "1","2","3":
  message (a);
endfor
```

There is no difference between these two examples.

Within each loop iteration, the ⟨symbolic token⟩ becomes a freshly created local-only temporary alias of the current object in the ⟨for list⟩.

With this example:

```
for a := "1",2,(origin--cycle),d:
  show a;
endfor
```

the local `a` will in turn be interpreted as a known string, known numeric, known path, and the symbolic variable `d`.

If for some reason you need to access the existing symbolic token `a` from inside the loop, you have to use `quote a` as explained in the article about MetaPost definitions, like you would inside inside a macro definition body.

The iterator variables (`a` in the example) are essentially identical to formal arguments inside macro definitions. Iterator variables are read-only.

You can also start a loop using a numeric progression:

```
for a = 1 step 1 until 3:
  message (decimal a);
endfor
```

It should be obvious that in this case the three numerics from the ⟨progression⟩ all have to produce known values.

In most MetaPost macro packages there is a macro named `upto` available that is defined as `step 1 until`. This allows for more natural input:

```
for a = 1 upto 3:
  message (decimal a);
endfor
```

You can also do a loop over a list of suffixes:

```
vardef mymessage @# =
  message (decimal @#)
enddef;

forsuffixes a = 1, 2:
  mymessage.a;
endfor
```

This type of loop is very useful for (typically short) lists of 'familiar' suffixes. For example, it is used in the `plain.mp` definitions of `dotlabels` and `penlabels`. Again, see the the article about MetaPost definitions for a detailed description of what suffixes are.

If the number of possible loop iterations cannot be determined beforehand, you can start a loop with the keyword `forever`:

```
forever:
  message ("eternal");
endfor
```

Especially with `forever:` (but also with the other loop types), it is also useful to be able to abort a loop mid-iteration:

```
a = 0;
forever:
  message ("eternal");
  exitif a>10;
  a := a + 1;
endfor
```

MetaPost does not have a way to skip to the next iteration but still remain in the loop (like '`continue`' in the language `C`). If you need that functionality, you will have to enclose some (or all) of the loop body inside a conditional.

Finally, there is a way to loop over a picture's content:

```
for a within currentpicture:
  if stroked a: message "stroked"; fi
endfor
```

The explanation of `stroked` and friends was already done earlier in this article, and `picture`s are the subject of another article.

Final words

This was a rather short article, because there are not that many primitives that control program flow inside MetaPost. This may feel as an oversight in the language if you are used to languages with more elaborate structures like symbolic `switch` statements and generic list filters. But at the most basic level, *all* program flow is just a combination of conditionals and jumps. MetaPost's set of built-in operations may be small and low-level, but it is sufficient. And nothing stops you from defining more complex flow control commands on top of those built-in operations.

Here is one such example (like the definition of `upto` seen earlier), the definitions of `range` and `thru` from `plain.mp`, that allow you to use shortcut ranges inside lists of suffixes, like so:

```
labels(1, range 100 thru 124, 223)
```

These definitions internally use a loop to generate an explicit list of suffixes for the outer `labels` command to use.

To end this article, here is another very small but useful definition from `plain.mp`:

```
def exitunless expr c = exitif not c enddef;
```

Taco Hoekwater

Colors and pictures

Abstract

This article is about MetaPost output. MetaPost produces graphics by means of `picture` variables that can contain a few different object types. The most important drawing object types can be colorized, so the first part of this article will talk about color data structures.

Colors

Color models

In order to understand how MetaPost handles color, it is necessary to understand a little bit about color models. Explaining that prerequisite knowledge in this article would make it much too long, so the assumption is made that you at least understand the difference between the basic principle behind greyscale, RGB, and CMYK specifications as ways to describe colors.

For once, this section does not start with a formal syntax. The formal specification would not really help because almost all the information we need cannot be seen in the expression syntax: parsing colors is easy for MetaPost. The interpretation that needs to happen after the reading has been done is the complicated bit.

MetaPost internally has four color models, any one of which can be chosen to do actual output with. Each of the color models also has an associated data type that can be used to define variables with that color model as its 'type':

- □ No model: `boolean`
- □ Greyscale: `numeric`
- □ RGB: `rgbcolor` (this is the initial default color model)
- □ CMYK: `cmykcolor`

None of these color models have an alpha/opacity component.

There is an internal variable `defaultcolormodel` that allows you to set a default color model:

```
defaultcolormodel := 5; % RGB
```

Each of the four color models MetaPost supports has an integer value associated with it, and these are the numerics used with `defaultcolormodel`. The numbers are: No model: 1, Greyscale: 3, RGB: 5, and CMYK: 7.
In case you are wondering: they are all odd because MetaPost uses the values 0, 2, 4, and 6 internally to signify ⟨unknown⟩ variables in each of these color models.

When you ask MetaPost to create a graphic element (a path or picture, as will be discussed in the next section) there are primitive operations to specify both the color model and the color value that is to be used while adding this object to the picture it will become part of.

The next set of examples use `draw` as example of creating a graphic element. All of the examples produce output with the color value 'black', depending on how that is done within that particular color model. The examples use `draw` as an educational shortcut, but in reality they apply to one of the primitive operations that will be discussed in the next section. A typical definition of the `draw` macro does more work than just a single primitive operation, so please focus on the color differences between the examples only.

First up, there is the option to use the current default:

```
draw p;
```

which uses a suitable 'black' definition for the current `defaultcolormodel`.

To use an explicit black greyscale when drawing a path:

```
draw p withgreyscale 0;
% or its alias:
draw p withcolor 0;
```

To use an explicit black RGB when drawing a path:

```
draw p withrgbcolor (0,0,0);
% or its alias:
draw p withcolor (0,0,0);
```

To use an explicit black CMYK when drawing a path:

```
draw p withcmykcolor (0,0,0,1);
% or its alias:
draw p withcolor (0,0,0,1);
```

From the above, you can see that `withcolor` is smart about what argument it gets and automatically picks the correct color model based on that value's specification. It will do the same thing if the value is a named variable. Use of the `withcolor` alias is recommended because it is shorter and (when used with color variables instead of literal values) it allows you to switch to a different color model without having to manually change every drawing command.

You will probably have noticed that the preceding examples covered only three of the four color models. The 'No color' color mode needs a bit more explanation.

The equivalent of

```
draw p;
```

is this:

```
draw p withcolor true;
```

Which uses the 'No model' color model to explicitly enable the black initialization. On its own that is not valuable. The real reason for the 'No model' is seen when the color model is used as a negation.

To skip black initialization when drawing a path, you can do this:

```
draw p withoutcolor;
% or its alias:
draw p withcolor false;
```

In this situation, the current object (`p`) will have no color information attached to it at all. No default 'black' will be output, so this object will be drawn with the color of the preceding object, if there is one. Be warned though that if this is the very first object to be output, it is likely it will still come out as black because usually printing systems start by initializing a default black color value.

Color variables

Variables of all color model types can be created using:

```
boolean mynocolor;
numeric mygreycolor;
rgbcolor myrgbcolor;
cmykcolor mycmykcolor;
```

As we saw earlier in the literal color model syntax examples, the input syntax for `rgbcolor` is a triplet of ⟨numeric expression⟩s inside parentheses, and for `cmykcolor` it is a quartet of ⟨numeric expression⟩s.

Just like `pair`s have `xpart` and `ypart` to access the parts of the variable, there are dedicated primitives for the RGB and CMYK color model parts as well.

For RGB:

```
redpart myrgbcolor;
greenpart myrgbcolor;
bluepart myrgbcolor;
```

For CMYK:

```
cyanpart mycmykcolor;
magentapart mycmykcolor;
yellowpart mycmykcolor;
blackpart mycmykcolor;
```

For orthogonality, there is also a primitive for the single greyscale part of a ⟨numeric⟩:

```
greypart mygreycolor;
```

These eight primitives can be used in equations, just like their `pair` counterparts.

The ⟨numeric expression⟩s that are used for the color parts in colors are treated a bit special when they are used as part of one of the primitive drawing commands. Most importantly, no error is produced when any of the parts are unknown or outside of the $[0, 1]$ range. They are just silently clipped to fit within the range. It is your responsibility as programmer to make sure that all the combination of ⟨numeric expression⟩s actually make sense as a color value.

Operations on colors

There are relatively few operations MetaPost can perform on RGB or CMYK colors as a singular object. Quite a lot of operations can already be done by manipulating the separate parts that were mentioned in the previous section, so there is little need for color-specific operators. Still, there are a few operations at ‘top level’ available.

You can multiply or divide color variables by a numeric:

```
rgbcolor myrgb;
myrgb = (0.5,0.5,0.5) * 1.5;
% => (0.75,0.75,0.75)
```

Or you can add or subtract two colors of the same type:

```
rgbcolor myrgb;
myrgb = (0.5,0.5,0.5) + (0.25,0.25,0.25);
% => (0.75,0.75,0.75)
```

You can also find the 'along-the-way' between two colors:

```
rgbcolor myrgb;
myrgb = .5[(0.5,0.5,0.5),(0.25,0.25,0.25)];
% => (0.375,0.375,0.375)
```

And colors can be negated:

```
rgbcolor myrgb;
myrgb = -(0.5,0.5,0.5);
% => (-0.5,-0.5,-0.5)
```

Finally, you can compare colors of the same type with each other:

```
rgbcolor myrgb, myrgba;
myrgb = (0.5,0.5,0.5);
myrgba = (0.25,0.25,0.25);
if myrgb > myrgba:
  message "true";
fi
```

Such tests process each component in order, and stop as soon as they notice a difference.

Operations on color variables like these may seem a bit useless at first glance, but the MetaPost macro packages that do three-dimensional drawings typically depend on color-based triplets or quartets as their data structures for points in space.

Pictures

MetaPost uses ⟨picture⟩s to internally store and eventually output graphical items. Here is the syntax tree for specifying ⟨picture⟩s:

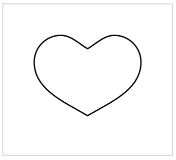
```
⟨picture primary⟩ → ⟨picture variable⟩
  | nullpicture
  | (⟨picture expression⟩)

⟨picture secondary⟩ → ⟨picture primary⟩
  | ⟨picture secondary⟩⟨transformer⟩

⟨picture tertiary⟩ → ⟨picture secondary⟩

⟨picture expression⟩ → ⟨picture tertiary⟩

⟨addto command⟩ → addto⟨picture variable⟩also⟨picture expression⟩⟨option list⟩
  | addto⟨picture variable⟩contour⟨path expression⟩⟨option list⟩
  | addto⟨picture variable⟩doublepath⟨path expression⟩⟨option list⟩

⟨option list⟩ → ⟨empty⟩ | ⟨drawing option⟩⟨option list⟩

⟨drawing option⟩ → withcolor⟨color expression⟩
  | withrgbcolor⟨rgbcolor expression⟩
  | withcmykcolor⟨cmykcolor expression⟩
  | withgreyscale⟨numeric expression⟩
  | withoutcolor
  | withprescript⟨string expression⟩
  | withpostscript⟨string expression⟩
  | withpen⟨pen expression⟩
  | dashed⟨picture expression⟩
```

There are simple parts like ⟨transformer⟩ and subexpressions in parentheses that can be skipped because we have talked about those before. The expression part of this syntax diagram is quite unremarkable, except for mentioning the one predefined picture variable: nullpicture. But this is a really important variable, because nullpicture is the way to create or reset a picture variable to the ⟨known⟩ (and empty) state.

Creating pictures and adding to them

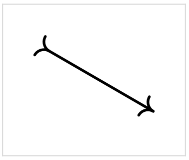
Before you look at the examples below, there is some information you should know. The examples use a predefined path with the name heart, like this:

```
path heart;
heart := (0,0){dir 30}..{up}(20,20)..
        {left}(10,30)..
        {dir -150}(0,25){dir 150}..
        {left}(-10,30)..{down}(-20,20)..
        {dir -30}cycle;
```



and a picture arrow_pic that already contains a simple graphic. The complete definition for that picture is:

```
path t_,h_,a_;
picture arrow;
arrow := nullpicture;
t_ := (-4,19){down}..{right}(0,15)
      {left}..{down}(-4,11);
h_ := t_ shifted (45,0);
a_ := (0,15)--(45,15);
def stroke_ =
  withpen pencircle scaled 1
enddef;
addto arrow doublepath t_ stroke_;
addto arrow doublepath a_ stroke_;
addto arrow doublepath h_ stroke_;
arrow := arrow rotated -30
               shifted (-25,12);
```



The last line of an example is typically

```
shipout A;
```

We will talk about the shipout command (and the withpen option) a bit later in the article. Now let's start with the examples ...

You create a new picture variable using picture:

```
picture A;
```

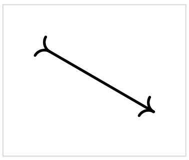
however, this creates a picture variable with the 'unknown' state. To convert it to a usable state, you *always* have to initialize it from another picture, for example:

```
picture A;
A = nullpicture;
```

Use an assignment (:=) instead of an equation (=) if you need to clear the receiving picture.

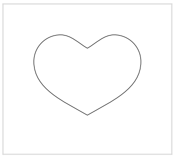
You can add another picture to a picture:

```
picture A,B;
A = nullpicture;
B = arrow;
addto A also B;
shipout A;
```



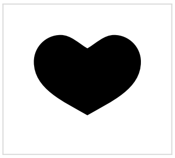
Add a stroked path to a picture:

```
picture A;
A = nullpicture;
addto A doublepath heart;
shipout A;
```



Add a filled path to a picture:

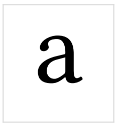
```
picture A;
A = nullpicture;
addto A contour heart;
shipout A;
```



The path must be cyclic for `contour` to work, because it needs a closed path to fill.

Adding a text label to a picture:

```
picture A,B;
A = nullpicture;
B = "a" infont "cmr10" scaled 4;
addto A also B;
shipout A;
```

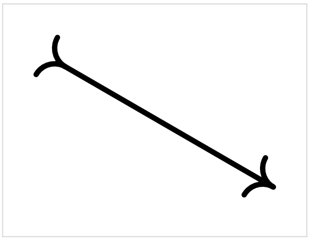


The `infont` operation is a bit special because it literally creates a picture and therefore it wants to be paired with a non-initialized picture variable. There is no need to assign `nullpicture` to `B` first. In fact, if `B` is a 'known' variable at this point, you will get the `Redundant or inconsistent equation.` error.

All three of the ⟨addto command⟩ versions accept a list of options that we will discuss shortly.

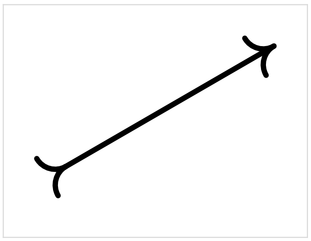
At the expression level, a picture expression can be transformed in all the normal ways:

```
picture A,B;
A = nullpicture;
B = arrow;
A := B scaled 2;
shipout A;
```



or (for example)

```
picture A,B;
A = nullpicture;
B = arrow;
addto A also B scaled 2 rotated 60;
shipout A;
```



Options to the addto command

The addto command forms accept various options.

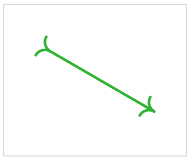
We have already encountered the color options:

withcolor⟨color expression⟩
withrgbcolor⟨rgbcolor expression⟩
withcmykcolor⟨cmykcolor expression⟩
withgreyscale⟨numeric expression⟩
withoutcolor

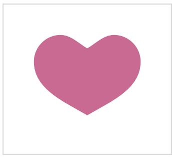
It is useful to know that when multiple color options are specified, the last one in the sequence 'wins'.

Here are a few examples:

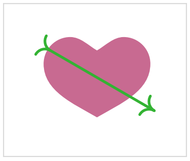
```
picture A,B;
A = nullpicture;
B = arrow;
addto A also B
  withrgbcolor (0.2, 0.7, 0.2);
shipout A;
```



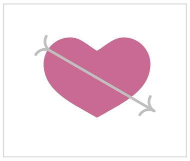
```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0);
shipout A;
```



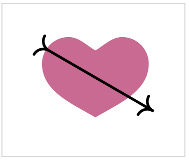
```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0);
addto A also arrow
  withcolor (0.2, 0.7, 0.2);
shipout A;
```



```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0);
addto A also arrow
  withgreyscale 0.75;
shipout A;
```

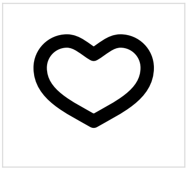


```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0);
addto A also arrow;
shipout A;
```



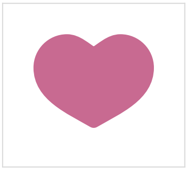
The withpen option allows specifying a pen:

```
picture A;
A = nullpicture;
addto A doublepath heart
  withpen pencircle scaled 5;
shipout A;
```



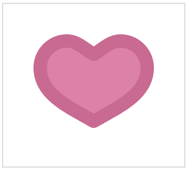
This also works for the contour case, where it then does ‘filldraw’:

```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.2, 0.7, 0.2, 0)
  withpen pencircle scaled 5;
shipout A;
```



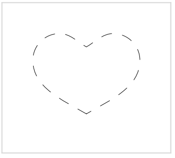
The example above uses a single color for both the filling and the stroking. If you want to use separate colors for each, you have to add two items to the image:

```
picture A;
A = nullpicture;
addto A contour heart
  withcmykcolor (0.1, 0.6, 0.1, 0);
addto A doublepath heart
  withcmykcolor (0.2, 0.7, 0.2, 0)
  withpen pencircle scaled 5;
shipout A;
```



With dashed, it is possible to specify a dash pattern to use for stroking a path. This accepts a picture as argument, so that needs be exist first. One of the simplest examples looks like this:

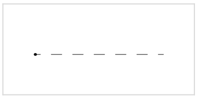
```
picture A,B;
A = nullpicture;
B = nullpicture;
addto B doublepath (0,0)--(2,0);
addto B doublepath (6,0)--(8,0);
addto A doublepath heart
  dashed B;
shipout A;
```



Dash patterns are quite special pictures. When the dash pattern gets used, MetaPost flattens whatever the content of the picture is onto the x axis. The left-most and right-most x values define the bounds of the pattern. The set of produced x values will then be used as the pattern to use to stroke the path the dash pattern is applied to. MetaPost will repeat that whole picture as a pattern if needed, but it will initially start at $x = 0$. This allows shifting of the pattern.

Here are the dashes from the example above again, but now applied to a straight line. I also added a dot to show to `(0,0)` point:

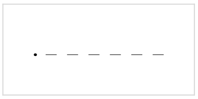
```
picture A,B;
A = nullpicture;
B = nullpicture;
addto A doublepath (0,0)
  withpen pencircle;
addto B doublepath (0,0)--(2,0);
addto B doublepath (6,0)--(8,0);
addto A doublepath (0,0)--(48,0)
  dashed B;
shipout A;
```



Note that the dash picture produces a repeating pattern 2 units on, 4 units off, 2 units on. The middle dashes in the example output are the same width as the gaps because they consist of the last part of the first repetition and the first part of the second repetition (and that repeated five times).

Shifting the pattern to the left by two units allows it to start with a gap.

```
picture A,B;
A = nullpicture;
B = nullpicture;
addto A doublepath (0,0)
  withpen pencircle;
addto B doublepath (0,0)--(2,0);
addto B doublepath (6,0)--(8,0);
addto A doublepath (0,0)--(48,0)
  dashed (B shifted (-2,0));
shipout A;
```



There are lots of rules for dash patterns because MetaPost typically uses primitive support in the backend to handle the actual dashing (e.g. `setdash` for Encapsulated PostScript output):

- ☐ A dash pattern should not contain text or filled objects (so only non-cyclic paths are allowed)
- ☐ None of the paths may overlap when projected on the x axis (and all the y coordinates are ignored)
- ☐ Any used pens (`withpen`) are ignored.
- ☐ Color settings (`withcolor` c.s.) are simply not allowed at all.

The last two limitations come from the fact that a dash patterns uses the pen and color of the object they are applied to. Finally, `dashed` does not work well with pens other than pens derived from `pencircle`. Again, this is because of limitations in the backend(s).

The final two options are for specifying pre- or postscripts:

```
withprescript⟨string expression⟩
withpostscript⟨string expression⟩
```

These can be useful when generating EPS or SVG output. It is not possible to give an actual example inside this (ConTEXt-processed) article, because ConTEXt uses these primitives for its own purposes, unfortunately.

But here is a listing of an example that assumes the default EPS output mode in standalone MetaPost:

```
picture A;
A = nullpicture;
addto A doublepath (0,0)
  withprescript "start1"
  withprescript "start2"
  withpostscript "stop1"
  withpostscript "stop2";
shipout A;
end.
```

When the above is processed by MetaPost, it will create an output file containing the typical EPS preamble followed by:

```
start2
start1
 0 0 0 setrgbcolor 0 0 dtransform truncate idtransform setline...
newpath 0 0 moveto 0 0 rlineto stroke
stop1
stop2
showpage
```

The `withprescript` and `withpostscript` options are therefore a lot like `special` in TEX: if you are familiar with PostScript (or SVG, for that output format), you can use these options to tweak the output to support features that are not possible within MetaPost itself, like for example spot colors or transparency.

Two uses of each option are included in the example to show off the relative ordering in the output when either one of them is specified more than once.

Picture commands

Possibly the most important command that can be used with a picture is `shipout`, because that instructs MetaPost to open an output file for the picture and convert its contents to the correct format. Using the command itself is simple:

```
shipout A;
```

This uses the internal variables `outputformat` and `outputtemplate` to construct the filename to be used.

There is a command to clip a picture to a path:

```
picture A;
path p;
...
clip A to p;
```

This path can have any shape, but it must be cyclic.

Set the bounding box of a picture to a path:

```
picture A;
path p;
...
setbounds A to p;
```

the path must be cyclic, and is always simplified to a rectangle based on the smallest and largest x and y values of the path's explicit points.

You can ask for the corners of a picture:

```
picture A;
pair t;
...
t = llcorner A;
% also lrcorner, urcorner, ulcorner
```

Finally, it is possible to loop over de contents of a picture using the `within` operator.

Using the `for` ... `within` operation, it is possible to ask for the constituent parts of each of the drawing items in a picture. The part names are given in a condensed form in the following examples. By recombining the extracted parts, it is possible to completely reconstruct a picture.

For these tests, you may have to check the type with an `if` test (one of `filled`, `stroked`, `clipped`, `bounded`, `textual`, as discussed in the article about conditionals) first, because not all graphical objects have all parts.

Here is the list:

Pre- and postscripts:

```
string part;
for v within A:
  part := prescriptpart v;
% postscriptpart
endfor
```

Transformation parts:

```
numeric part;
for v within A:
  part := xpart v;
% ypart xxpart yypart xypart yxpart
endfor
```

Color model and/or color part

```
numeric part;
for v within A:
  part := colormodel v;
endfor
```

Color parts (RGB)

```
numeric part;
for v within A:
  part := redpart v;
  % bluepart greenpart
endfor
```

Color parts (CMYK)

```
numeric part;
for v within A:
  part := cyanpart v;
  % magentapart yellowpart blackpart
endfor
```

Color parts (grey)

```
numeric part;
for v within A:
  part := greypart v;
endfor
```

The dash part (which is itself a picture):

```
picture part;
for v within A:
  part := dashpart v;
endfor
```

The pen part:

```
pen part;
for v within A:
  part := penpart v;
endfor
```

The path part:

```
path part;
for v within A:
  part := pathpart v;
endfor
```

The text part of a label:

```
string part;
for v within A:
  part := textpart v;
endfor
```

The font part of a label:

```
string part;
for v within A:
  part := fontpart v;
endfor
```

Summary

That wraps up this article about the primitive operations on `pictures` and colors. As usual, many of the commands mentioned here are normally hidden behind macro definitions. In particular, as far as I know all of the MetaPost macro packages define a macro `draw` for adding stroked paths and a macro `fill` for adding filled paths to a picture. These are then used in combination with a predefined picture variable called `currentpicture`. Macros packages usually predefine the primary RGB colors `red`, `green`, and `blue` as well.

Maybe more higher level commands are available. For that, you will have to check the documentation of the macro package you are using.

Taco Hoekwater