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