MetaPost Definitions

Taco Hoekwater

Definitions in MetaPost are a fairly complicated subject. This talk and paper 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 last year's 'Sparks, Tags, Suffixes and Subscripts' article.

1. 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 5. 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.

1.1 `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.

1.2 `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`.

1.3 `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 `()`).

1.4 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.

1.4 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.

1.5 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
```

(⟨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 63 for the explanation of ⟨expression⟩ and ⟨primary⟩). It is not possible to fake the `point of` primitive syntax in another way.

2. 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' ...

2.1 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 2.2 for how to define binary operators for the top three levels. Primary operators in MetaPost are always unary operators.

2.2 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`.

3. 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`:

⟨macro definition⟩ → ⟨macro heading⟩ = ⟨replacement text⟩ `enddef`
⟨macro heading⟩ → `vardef` ⟨declared variable⟩ ⟨delimited part⟩ ⟨undelimited part⟩
| `vardef` ⟨declared variable⟩ `@#` ⟨delimited part⟩ ⟨undelimited part⟩

The ⟨delimited part⟩ and ⟨undelimited part⟩ are the same as before and are not repeated.

The use of ⟨declared variable⟩ instead of the ⟨symbolic token⟩ from the earlier definition commands is important: This is what makes this type of definition produce a **tag** instead of a **spark**. The ⟨declared variable⟩ 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 ⟨macro heading⟩ 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.

4. Grouping

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

⟨statement⟩ → ⟨equation⟩ | ⟨assignment⟩ | ⟨declaration⟩
| ⟨definition⟩ | ⟨title⟩ | ⟨command⟩ | ⟨empty⟩
| `begingroup` ⟨statement list⟩ ⟨statement⟩ `endgroup`

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.

5. 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. 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.

6. 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`

7. 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.