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⟩

⟨boolean expression⟩ → ⟨boolean tertiary⟩
| ⟨numeric expression⟩⟨relation⟩⟨numeric tertiary⟩
| ⟨pair expression⟩⟨relation⟩⟨pair tertiary⟩
| ⟨transform expression⟩⟨relation⟩⟨transform tertiary⟩
| ⟨boolean expression⟩⟨relation⟩⟨boolean tertiary⟩
| ⟨string expression⟩⟨relation⟩⟨string tertiary⟩

⟨relation⟩ → < | <= | > | >= | = | <>

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 `pictures` 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