

METAPOST 的艺术

何 力

demonstrate@163.com

2006 年 2 月 7 日

摘要

METAPOST 是一种解释性作图语言，它的输出就是另外一种作图语言。了解它，不仅仅方便你在 $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 中用图形表达你的思想，还能够制作出各种各样精美的图片，动画。

目录

目录	1
插图	3
表格	3
 第一部分 理论篇	 4
1 METAPOST 的历史	5
1.1 我的 METAPOST 在哪里？	6
1.2 使用 METAPOST 的基本流程	6
1.3 一个简单的例子	7
1.4 METAPOST 的预览问题	7
2 变量类型	8
2.1 numeric 类型	8
2.2 pair 类型	9
2.3 color 类型	10
2.4 path 类型	11
2.5 pen 类型	14
2.6 string 类型	14
2.7 picture 类型	15
2.8 transform 类型	16
2.9 数组	17

目录	2
3 控制结构	19
3.1 条件控制	19
3.2 循环控制	19
3.3 自定义宏	20
3.3.1 def	20
3.3.2 grouping	20
3.3.3 expr、text 和 suffix	21
3.3.4 vardef	22
3.3.5 算符	23
4 作图基本知识	27
4.1 选择什么样的笔?	27
4.2 线型	27
4.3 端点	30
4.4 接头	30
4.5 箭头	32
第二部分 实践篇	35
5 FAQ	36
5.1 字体显示不对或者不能显示	36
5.2 合适的 editor	36
5.3 如何在 PDF \LaTeX 中使用 METAPOST?	36
6 调试	37
7 常用宏	38
8 常见宏包简介	39
第三部分 综合篇	40
9 使用 METAPOST 制作精美动画	41
10 Gallery	42
11 C++ flavor: Asymptote	43
第四部分 附录	44
A 贡献者	45
B 已知问题	46
参考文献	47

插图	3
索引	48

插图

1	一个简单的 METAPOST 示意	7
2	单位其实就是一种全局变量	8
3	变量做单位用	9
4	一个二元正态分布的例子	9
5	三角形的重心	10
6	使用颜色	11
7	直线和曲线	11
8	控制点的作用	12
9	通过的角度计算	12
10	使用 dir 控制曲线形状	13
11	... 与 .. 的不同	13
12	不同 tension 的影响	13
13	不同 curl 的曲线形状	14
14	自定义 pen	14
15	一把直尺的刻度	15
16	infont 创建的字体	15
17	不共线三点确定 transform	16
18	Cantor 集合	17
19	if 示意	19
20	三叶玫瑰线	20
21	一个 grouping 示例	21
22	save 与 interim	22
23	primary vs expr	24
24	METAPOST 中的词法	25
25	算符优先顺序	26
26	不同线型作图	28
27	线型研究	29
28	dashed 怎么利用线型	29
29	端点样式	30
30	接头样式	31
31	miterlimit 的影响	32
32	箭头示意	33
33	自定义箭头	34

表格

1	addto 的等价用法	16
---	--------------------	----

第一部分 理论篇

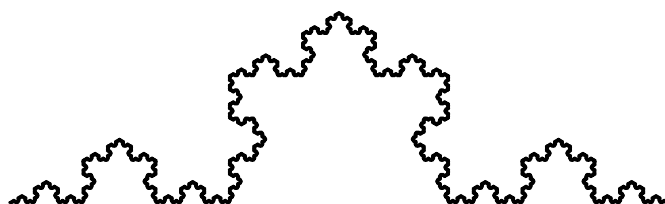
本部分讲述一些使用 METAPOST 作图的理论知识，从 METAPOST 的历史开始，然后讲述 METAPOST 使用的变量类型，以及其中使用的控制结构，最后讲述作图的一些基本概念和方法。

要点：

- 熟悉使用 METAPOST 作图以及插入到所需位置的基本流程。
- 理解 METAPOST 是一个宏语言的特征。
- 熟练的使用 METAPOST 的几种基本数据类型。
- 能够使用 METAPOST 的控制结构写出简单的程序。
- 能够运用所学会的编程知识看懂系统写的宏。
- 逐步积累一点点常用的宏。
- 能够理解并运用各种作图的基本手段实现自己的作图目的。

难点：

- 怎么正确的理解 METAPOST 是一个宏语言。
- 如何利用已有的方式构造合适的 path。
- 如何利用线性方程组简单的计算出所需要的未知数。
- 如何正确理解 `def` 和 `vardef` 的不同。
- 如何实现宏的后缀。
- 如何定义算符，以及不同级别算符的区别。
- 怎么理解线型以及定义自己的线型。



1 METAPOST 的历史

METAPOST 作为一门语言，其发生，发展，到今天有着自己的历史。大牛 Donald E. Knuth 为其巨著 [Knu] 的排版头痛从而创作了 \TeX 这个排版语言，同时为了为该排版系统定制字体，开发了 METAFONT¹。METAPOST 就是由 John D. Hobby 基于 METAFONT 创造，学习 METAPOST 应该参考他所作的 METAPOST 手册 [Hobc]，本文不希望写成如此的手册，而是希望通过一些基本概念方法的介绍，激发读者使用 METAPOST 的欲望。

METAFONT 输出的是点阵字型，而 METAPOST 所输出却是 POSTSCRIPT²。METAPOST 从 METAFONT 中借用了一些基本操纵图形的工具，也从 POSTSCRIPT 中获得了更多的新特性，如 clipping（剪辑），shading（渐变色），dashed lines（虚线）等等。

在网上可以找到很多与 METAPOST 相关的资料，下面列出少许供参考：

- <http://cm.bell-labs.com/who/hobby/MetaPost.html> 这是 John D. Hobby 的 METAPOST 官方主页了，上面有关于 METAPOST 的历史资料，如何安装 METAPOST，怎么利用 METAPOST 的例子，手册，mail list。
- <http://www.tug.org/metapost.html>，这是 TUG 的 METAPOST 介绍网页，有很多教程（在本文的参考手册中可以找到部分相同的），相关的应用，相关的作图语言（如 Asymptote）、程序。
- <http://tex.loria.fr/prod-graph/zoonekynd/metapost/metapost.html>，全是例子，如果你有耐心把这些程序全部写完，你马上就能成为一个 METAPOST 高手，这里面的例子从简单到困难，本文有些例子可能也会从中选择出来。
- <http://bbs.ctex.org/forums/index.php?showforum=35>，这是国内人气最旺的 \CTEX 论坛的 METAPOST 分坛，由 elove 老兄主持，有问题可以去问问，有好东西可以去分享，有经验可以去交流。
- <http://melusine.eu.org/syracuse/metapost/>，这是一个法文站点，上面除了一些文档以外（如 [Hobc] 的 prosper 版本）还含有大量的动画，漂亮的 galleries，使用 METAPOST 做成的课件，感谢 changroc 提供。另外该网站还有 POSTSCRIPT，PSTricks，SWF 的相关示例。

这里顺带着讲述一些作图的方法。在 Windows 大行其道的今天，很多人都被漂亮的 GUI 所迷惑，熟悉了怎么使用鼠标拖拖拉拉，反倒生疏了键盘。为了获得一些文档中的图片，想来一般人都尝试过 Windows 的画笔，甚至是 MS Office 中的作图工具，另外微软还提供了 MS Visio。这些作图工具的特点在于用户不必关心准确的位置，只需要到一些面板上寻找按钮到画布上拖拉，就能获得一些图形，Visio 稍微好点带有标尺，可以较为精确的定位，同时 Visio 的输出可以选择矢量图形格式，加上大量的图形元件，可以很方便的实现多种图形的绘制。另外，微软在 MS Office 2003 中捣鼓出来了自己的 mdi 格式，大家可以猜测微软的目的。在 Windows 平台下，有不少类似的作图工具，如 Adobe 的平面设计“专家”Photoshop，矢量图形绘制 illustrator，还有 Corel 的点阵矢量包打的 Corel Draw，另外和 Visio 有的一拼的还有 SmartDraw 等等。如果算上开源作品中精良的 GIMP 和 Inkscape，那真是百家齐放。Linux 下面还有很多类似的软件，也有移植到 Windows 下的，如 xfig、dia 等等，Open Office 里面的 oodraw 和 KDE Office 套装中的 kivio，也算是不错的 Visio 替代品。

可是有人能做到像 AutoDesk 的 AutoCAD 那样精准的图形么？“哦，”你也许会抱怨道，“我不需要这种图形。”那么这里不妨劝你放弃 METAPOST，因为 METAPOST 要处理获得的往往是精确的图形，而随意画出来的图形却很难用 METAPOST 画出来。另外，你不能指望可以仅仅用鼠标就让

¹METAFONTTM 是 Addison Wesley Publishing 公司的注册商标。

²POSTSCRIPTTM 是 Adobe System Inc. 的注册商标。

METAPOST 服服贴贴的为你画出各种图形，正如 AutoCAD 一样，真正的高手很少完全用鼠标，并且结合 Visual LISP 有时才能高效的绘制出所需要的图形。“哦，METAPOST 这么繁啊！我不想用了……”好吧，这算是我对你打的预防针，因为的确学习 METAPOST 你需要更大的热情和精力，看看上面那个全部是例子的网页吧，难道精美的图片不值得我们热情和精力么？

当你下定决心时，就让我们开始 METAPOST 之旅吧！

1.1 我的 METAPOST 在哪里？

如果你在 Windows 下面，安装了 \TeX 套装，或者你是独立的安装了 Mik \TeX 发行版，那么你可以用 Mik \TeX 的 Package Manager 查看自己的 METAPOST 是否已经搞定，没有的话可以在其中完成安装。 \TeX 套装的 full 版用户，如果你没有删除过 METAPOST，你就可以直接使用 METAPOST 了。

如果你在 Linux 下面，安装的是 tetex，一般说来 METAPOST 已经装好了，你可以用

```
$ whereis mpost
```

或者

```
$ whereis mp
```

看看该程序是否在你的 PATH 中。如果没有安装，可以依照各自发行版安装对应的 tetex，如 Fedora Core 用户可以考虑使用

```
# yum install tetex tetex-fonts tetex-latex tetex-xdvi \
    tetex-afm tetex-doc tetex-dvips
```

安装好整个 tetex，想单另装可以去 CTAN 下载源代码自己编译，Debian 用户可以使用 aptitude 这个 TUI 或者

```
# apt-get install tetex-base tetex-bin tetex-doc \
    tetex-extra
```

从网上安装。

现在我使用的 Mik \TeX 版本和 tetex 的版本分别为 2.41 和 3.0，里面所带的 METAPOST 的版本为 0.641。

1.2 使用 METAPOST 的基本流程

一般说来，最好先对需要做的图形有个基本的轮廓，然后开始编写源代码，写好后使用 mpost 程序将源程序转换成为 POSTSCRIPT，可以使用一般的查看软件如 GSView 或者 ImageMagick/-GraphicMagick 自带的 display/gm display 查看。如果不符合自己的要求，就修改源代码，重新转换，直到生成自己需要的图形为止。该图片可以使用 graphics/graphicx 宏包直接插入到 \LaTeX 源文件中，然后编译 \LaTeX 源文件获得 .dvi 文件，使用 DVI 阅读软件，如 Mik \TeX 的 yap 和 Linux 下的 xdvi(k)。 \TeX 用户可以通过

```
\input epsf
% something
$$ \epsfbox{simple-example.0} $$
```


插入图片。

另外可以使用 mptopdf 将 .mp 直接转化为 PDF，如上面的例子就会被转化为 simple-example-0.pdf。

1.3 一个简单的例子


那么首先让我们建立一个小小的测试文件（参考图1）。存为 `simple-example.mp`，并使用

图 1: 一个简单的 METAPOST 示意

	<pre> beginfig(0) draw (0, 0) — (1cm, 0) — (1cm, 1cm) — (0, 1cm) — cycle ; endfig ; end </pre>
---	---

```
$ mpost simple-example.mp
```

获得编译结果，是一个名为 `simple-example.0` 的文件，你可以检查是不是和在本文档中所见一致，是一个边长为 1cm 的正方形。

 你能猜出来上面的程序大致什么意思么？

1.4 METAPOST 的预览问题


从上面的例子看出，如果希望能够在写出代码后较快的看到转化的结果，需要用一道程序实现预览，这在 WinEdt 里面已经实现过了，注意工具栏靠右侧的两个按钮，有一个是用来配置预览，还有一个就是用于预览，其基本原理就是将生成的图形建立在一个临时的 `.tex` 文件中并且编译后使用 `dvips` 转换为 `POSTSCRIPT` 并使用 `GSView` 查看。

在 Linux 里面，可以配合一些脚本实现类似的功能，如下面的一段 BASH 脚本利用 `find` 实现类似的预览功能：

```

#!/bin/sh
if test $# -gt 1
then
  fn=$1
  shift
  options=$*
else
  fn=$1
  options=""
fi
echo "mpost_$options_\\"prologues:=1; input ${fn}.mp\"
mpost $options "\prologues:=1;_input_${fn}.mp"
find -name "${fn}.*[0-9]*" -exec display {} \;

```

 你能否想出更好的预览脚本？或者编制一个自己的预览程序？

2 变量类型

值得注意的几点是 METAPOST 程序的风格，这里提出几点：1. 使用 `;` 分割不同句子。2. 使用 `%` 作为行注释号。3. 使用 `:=` 作为赋值号。


2.1 numeric 类型

由于 METAPOST 不需要像 C/C++ 一样使用很大范围的整数或者浮点数，因此 METAPOST 选择使用的 numeric 类型是不分整数浮点，其表达精度是 $\epsilon := \frac{1}{65535}$ ，表达的范围在 $(-4096, 4096)$ 之间。之所以出现 4096 是因为这是对对应到 POSTSCRIPT 中的距离已经超过 1.4m 了，一般不需要更大的数了，由此可见 METAPOST 使用的固定精度（小数点后 16 位），另有 12 位表达整数部分，还有一位符号位。可以在计算数值超过限定范围后，把一个内置变量 `warningcheck := 0`；即可。另外，numeric 变量不需要声明即可使用。

一般说来，numeric 类型被用于表征长度，默认的单位是 $\text{bp} = \frac{1}{72}\text{in}$ 。但是 METAPOST 也支持其他很多常见单位，如 `cm`, `mm`, `pt` $= \frac{1}{72.27}\text{in}$, `cc` $= 12.97213\text{bp}$, `dd` $= 1.06601\text{bp}$, `pc` $= 11.95517\text{bp}$ 。


另外，这些单位和我们使用的 numeric 变量没有太大的区别，换言之，你完全可能会通过对某些单位的赋值，使得该单位失去原先的意义，但这往往是我们不希望看到的：当我们在某处改变了某个单位后，在该文件其他任何后继部分除非再次更改该值，否则都不会变化。下面的简单实验说明了这一点。

图 2: 单位其实就是一种全局变量

	<pre> beginfig(0) draw (0, 0) — (1cm, 0) ; cm := cm / 2 ; draw (1cm, 0) — (1cm, 1cm) ; endfig ; beginfig(1) draw (0, 0) — (1cm, 0) ; cm := cm * 2 ; draw (1cm, 0) — (1cm, 1cm) ; endfig ; end ; </pre>
---	--

可能你注意到了 `1cm` 的写法，既然变量和单位基本雷同，你当然可以通过自己定义一个长度，如 `u := 1cm`；然后通过 `5u` 这种类似的方法作图，最终调节 `u` 获得不同大小的图片，如图 3。

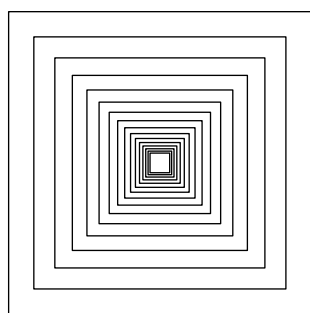
但是值得注意的是不同的 `beginfig ... endfig`；环境中的 `u` 是不互相影响的。

 请自己设计实验验证。

numeric 可以执行的运算类型有 `+`, `-`, `*`, `/`，另外还可以做 `**`, `++`, `+++`, `abs`, `mod`, `div`, `mexp`, `mlog` 这些运算，分别

$$\begin{aligned}
 x ** y &= x^y & x ++ y &= \sqrt{x^2 + y^2} \\
 x +-+ y &= \sqrt{x^2 - y^2} & \text{abs } x &= |x|
 \end{aligned}$$

图 3: 变量做单位用



```

beginfig( 0 )
  u := 1cm ;
  for i := 1 upto 16 :
    draw ( -2u, -2u ) -- ( 2u, -2u ) --
          ( 2u, 2u ) -- ( -2u, 2u ) -- cycle ;
    u := u / 1.2 ;
  endfor ;
endfig ;

end

```

$$x \operatorname{div} y = \left\lfloor \frac{x}{y} \right\rfloor$$

$$x \operatorname{mod} y = x - y \times (x \operatorname{div} y)$$

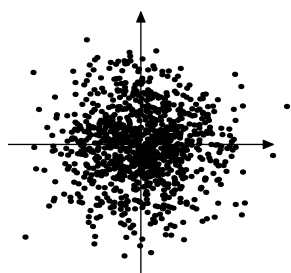
$$\operatorname{mexp} x = e^{\frac{x}{256}}$$

$$\operatorname{mlog} y = 256 \log y$$

2.2 pair 类型

所谓的 pair 类型也就是我们通常所指的平面直角坐标系中点的坐标了，在 METAPOST 里面和一般的数学表达完全相同，即 (x, y) ，其中的 x, y 都是 numeric 类型的变量。pair 型变量使用前必须声明。下面的例子说明如何产生一个二元正态分布，其数学期望在 $(0, 0)$ ，协方差矩阵为 $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ 。

图 4: 一个二元正态分布的例子



```

beginfig( 0 )
  u := 5mm ;
  drawarrow ( -3.5u, 0 ) -- ( 3.5u, 0 ) ;
  drawarrow ( 0, -3.5u ) -- ( 0, 3.5u ) ;
  pickup pencircle scaled 2pt ;
  for i := 1 upto 1000 :
    draw ( u * normaldeviate, u * normaldeviate ) ;
  endfor ;
endfig ;

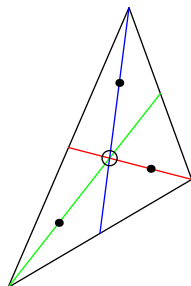
end

```

pair 与 pair 相加可认为向量相加，其内积使用 $x \operatorname{dotprod} y$ 表示，是一个 numeric 的结果。pair 与 numeric 相乘就相当于线性代数中的数乘运算。如 $a(x, y)$ 或者 $a * (x, y)$ ，表示的就是 $(a*x, a*y)$ 这个点。

pair 可以进行求定比分点的运算，其基本语法是 $a[x, y]$ ，其中 a 是一个 numeric 类型变量，而 x, y 皆为 pair 型变量，该表达式返回的是 $x + a(y - x)$ 这一点，因此 $a = \frac{k}{n}, k = 1, \dots, n-1$ ，就获得了 $n-1$ 个 n 等分点。图 5 的程序说明了怎么利用定比分点求证三角形的三条中线交于一点，并且这个点三等分中线。

图 5: 三角形的重心




```

beginfig( 0 )
  pair a, b, c, ab, bc, ca ; u := 5cm ;
  a := ( uniformdeviate( u ), uniformdeviate( u ) ) ;
  b := ( uniformdeviate( u ), uniformdeviate( u ) ) ;
  c := ( uniformdeviate( u ), uniformdeviate( u ) ) ;
  ab := .5[ a, b ] ; bc := .5[ b, c ] ; ca := .5[ c, a ] ;
  draw a — b — c — cycle ;
  draw a — bc withcolor ( 1, 0, 0 ) ;
  draw b — ca withcolor ( 0, 1, 0 ) ;
  draw c — ab withcolor ( 0, 0, 1 ) ;
  draw fullcircle scaled 2mm shifted 2/3[ a, bc ] ;
  pickup pencircle scaled 3pt ;
  draw 1/3[a, bc] ; draw 1/3[b, ca] ; draw 1/3[c, ab] ;
endfig ;

end

```

 我们知道三角形除了重心以外，还有垂心，内心，外心，旁心，如何设计类似的程序验证？垂心是三条高的交点，内心是三条角平分线的交点，外心是三条中位线的交点，旁心是一条内角平分线和两条外角平分线的交点。

我们可以使用 `xpart`, `ypart` 从一个 `pair` 中取出其某方向坐标。比如有一个 `pair` 为 `a`，那么和其正交的方向可以用 `(-ypart a, xpart a)` 表示。


可以使用下面的方法求两条直线的交点：因为交点在两条直线上，都可以看作其上两点的定比分点，只是对应的定比未知，所以可以列出一个二元一次方程组（未知数为两个定比）求解。但是 `METAPOST` 提供了更好的方法，

```

pair a, b, c, d, e ;
% initial for a, b, c, d
e = whatever[a, b] ;
e = whatever[c, d] ;

```

这里的 `a`, `b`, `c`, `d` 都是已知，`e` 是过 `a`, `b` 的直线与过 `c`, `d` 的直线的交点。注意到这里使用的 `=`，并非 `:=`，表示的正是一个线性方程组。而 `whatever` 是一个匿名 `numeric` 变量（每次调用代表的不同的变量），表示的正是方程组中代求的定比（虽然不是我们最终要求的交点）。

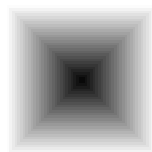
 请设计一个实例，使用自定义的 `numeric` 变量计算交点，看看结果是不是和 `whatever` 计算出来的交点相同。

其实利用 `=` 构造多元线性方程组同样适用于 `numeric` 类型，不妨写一个，结果可以用 `show variable_list` 显示，其中的变量表用 `,` 分割。当需要更加翔实的结果显示该命令的结果时，可以 `tracingonline := 1`；。

2.3 color 类型

每种颜色都是用红绿蓝三色依照不同比例组合而成，因此通常用三种颜色的比例来表示一种颜色，故 `(r, g, b)` 中 `r`, `g`, `b` 的值都是在 `[0, 1]` 中选择的，其中像 `black := (0, 0, 0)`, `white := (1, 1, 1)`, `red := (1, 0, 0)`, `green := (0, 1, 0)`, `blue := (0, 0, 1)`。

图 6: 使用颜色




```

beginfig( 0 )
  m := 1cm ;
  for i := round( m ) downto 1 :
    fill ( -i, -i ) -- ( -i, i ) -- ( i, i ) --
          ( i, -i ) -- cycle withcolor ( i/m*white ) ;
  endfor ;
endfig ;
end

```

颜色也可以加减，可以做数乘，还能算分比色，这都和 pair 类似。使用颜色可以参考图 6。

 考虑如何把上面的程序修改成为任意两种颜色之间颜色渐变的程序。(提示: 使用分比; 原问题由于一种颜色是黑色所以写得较简单)。

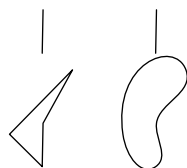
类似于 xpart, 这里有 redpart、greenpart、bluepart。

2.4 path 类型

其实在很多例子中我们已经使用到了一些简单的 path, 而且使用过程中你会觉得这是如此的简单, 甚至我即便没有特指其意义, 你也很快猜出来了。不错, 那些类似于 $a -- b$ 产生的东西就是 path (其中 a, b 是 pair), 而且这些是用一条线段连接 a, b 的 path。有时, 你会见到 **cycle** 出现在末尾, 它表示的就是这条 path 最开始的那个 pair, 所以你看得到这种 path 都是闭合的。

下面, 我们将介绍如何产生曲线。一个基本的手法就是把 $a -- b$ 换为 $a .. b$, 表示用尽量光滑的曲线连接, 图 7 例子说明了这种应用的结果。从图上可以看出, 如果只有两个点产生的曲线就是直线。

图 7: 直线和曲线



```

beginfig( 0 )
  u := 1.5cm ;
  pair p[] ;
  for i := 1 upto 4 :
    p[i] := ( uniformdeviate( u ), uniformdeviate( u ) ) ;
  endfor ;
  draw p1 -- p2 -- p3 -- p4 -- cycle ;
  draw ( p1 .. p2 .. p3 .. p4 .. cycle ) shifted ( u, 0 ) ;
  draw ( p1 -- p2 ) shifted ( 0, u ) ;
  draw ( p1 .. p2 ) shifted ( u, u ) ;
endfig ;

end

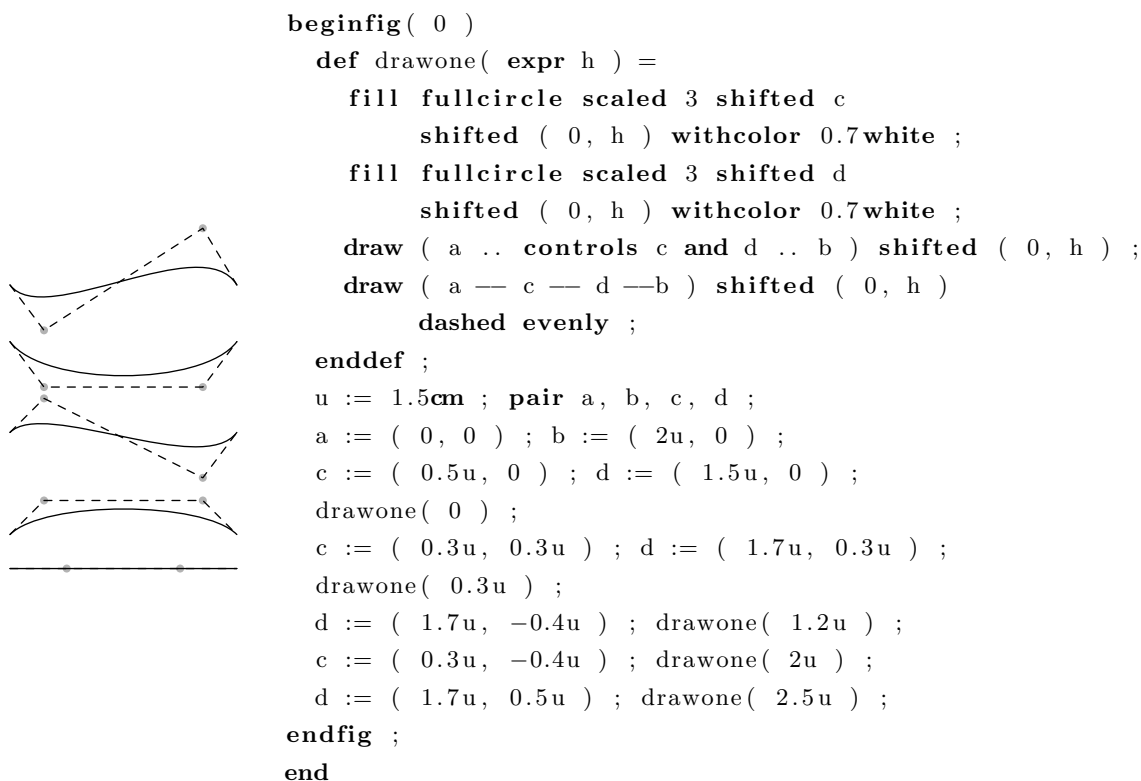
```

你可能会想, 这是基于什么原理产生的曲线呢? 答案是三次 Bézier 曲线, 如果你没有学过数值逼近这类型的课程, 也不要紧。你只要记住他能把即将通过的每一个点用光滑的 (至少两阶连续可导) 曲线连接起来。同时, 你可以把 $--$ 和 $..$ 混用在一起。

在一条 Bézier 曲线上任两个相邻的顶点之间的曲线段都是由两个控制点控制的, 如顶点是 a, b ,

指定控制点为 c, d , 当 a, c, d, b 形成一个凸的折线段时, 所形成的 Bézier 曲线在这条折线段的外侧, 我们可以用 `a .. controls c and d .. b` 声明这种结构。参考图 8。

图 8: 控制点的作用



可能直接控制这些点的位置并不能让你很直观的改变曲线的形状, 所以 METAPOST 还提供了通过指定一个曲线通过方向的参数, 如 `.. x{dir a} ..`, 其中 x 是一个顶点, `numeric` 类型的 a 是表示通过的角度, 计算方法如下图: 常用的方向也可以直接用单词代替, 如 `dir 0` 就是 `left`, `dir 90` 就是

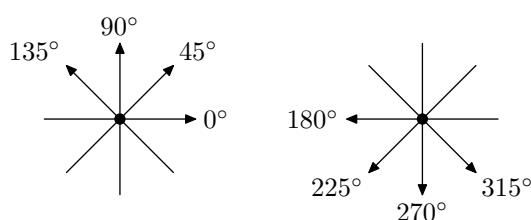


图 9: 通过的角度计算

`up`, `dir 180` 就是 `right`, `dir 270` 就是 `down`。图 10 这里有一个 [Hobc] 中的例子。

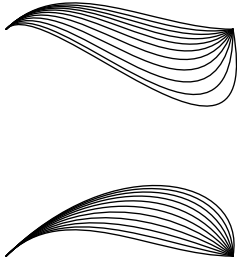


如果一个点两侧都设置了方向会有什么结果? 比如 `.. {right} origin {up} ..`。

取自 [Hobc] 上图 11 上的结果说明, 只指明方向是不能完全确定一条曲线的, ... 的引入就是为了画出尽可能在 inflection 不变下的曲线。

另外还可以用 `tension` 来控制曲线的形状, 它描述的是曲线的张紧程度, 默认值为 1, 图 12 说明了不同 `tension` 下曲线的不同形状。常见的用法是 `a..tension t..b` 或者 `a..tension s and t..b`, 这会设置连接 a, b 曲线的张紧程度。

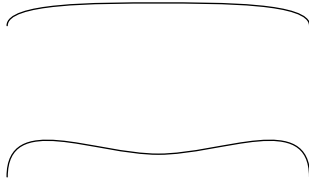
最后提到的一个方法是使用 `curl`, 其方式为 `a{curl c}..`, 看起来和 `dir` 类似。其意义在于设置了

图 10: 使用 `dir` 控制曲线形状

```

beginfig( 0 )
  u := 3cm ;
  for i := 0 upto 10 :
    draw ( 0, 0 ){dir 45} .. {dir -10i}( u, 0 ) ;
  endfor ;
  for i := 0 upto 10 :
    draw ( 0, u ){dir 45} .. {dir 10i}( u, u ) ;
  endfor ;
endfig ;
end

```

图 11: `...` 与 `..` 的不同

```

beginfig( 0 )
  u := 1cm ;
  pair a, b, c ;
  a := ( 0, 0 ) ;
  b := ( 2u, .3u ) ;
  c := ( 4u, 0 ) ;
  draw a{up} .. {right}b .. {down}c ;
  draw ( a{up} ... {right}b ... {down}c )
    shifted ( 0, 2u ) ;
endfig ;
end

```

图 12: 不同 `tension` 的影响

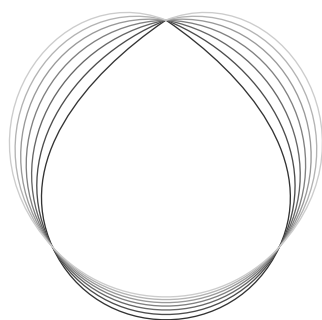
```

beginfig( 0 )
  u := 3cm ;
  pickup pencircle scaled 2 ;
  for i := -1 upto 3 :
    draw ( 0, 0 ) .. ( u, 0 ) .. tension ( 1 + 0.2i )
      .. ( u, u ) .. ( 0, u ) .. cycle
      withcolor ( (.3 + 0.2i)*white ) ;
  endfor ;
endfig ;
end

```

该处的曲率大小，默认为 1，越接近零，该处越小。图 13 展现了不同 curl 的控制结果。

图 13: 不同 curl 的曲线形状



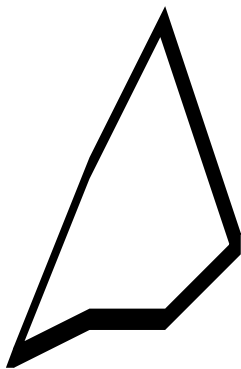
```
beginfig( 0 )
  u := 1.5cm ;
  for i := -3 upto 3 :
    draw ( 0, 0 ) .. {curl (0.3 i + 1)}( u, 2u )
      .. ( 2u, 0 ) .. cycle withcolor
        ( ( 0.1 i + 0.5 ) * white ) ;
  endfor ;
endfig ;
end
```

path 类型的变量可以用 a & b 粘合，如果 a 的终点和 b 的起点重合。

2.5 pen 类型

所谓的 pen 其实和 path 极为类似，两者甚至可以互相转换，如 **makepen**(closedpath) 返回一个 pen 类型，而 **makepath**(pen) 就把 pen 还原为 path 了。因此借助于不一样的 pen 我们可以获得不一样的作图效果，默认的 pen 是一个圆，也就是 **pencircle** 了。下面的例子中我们使用自己做出来的 pen 画几笔看看。

图 14: 自定义 pen



```
beginfig( 0 )
  u := 1cm ;
  pen mypen ;
  mypen := makepen( ( 0, 0 ) — ( 3, 0 ) —
    ( 3, 8 ) — cycle ) ;
  draw ( 0, 0 ) — ( u, 0 ) — ( 2u, u ) —
    ( u, 4u ) — ( 0, 2u ) — ( -u, -.5u ) —
    cycle withpen mypen ;
endfig ;
end
```

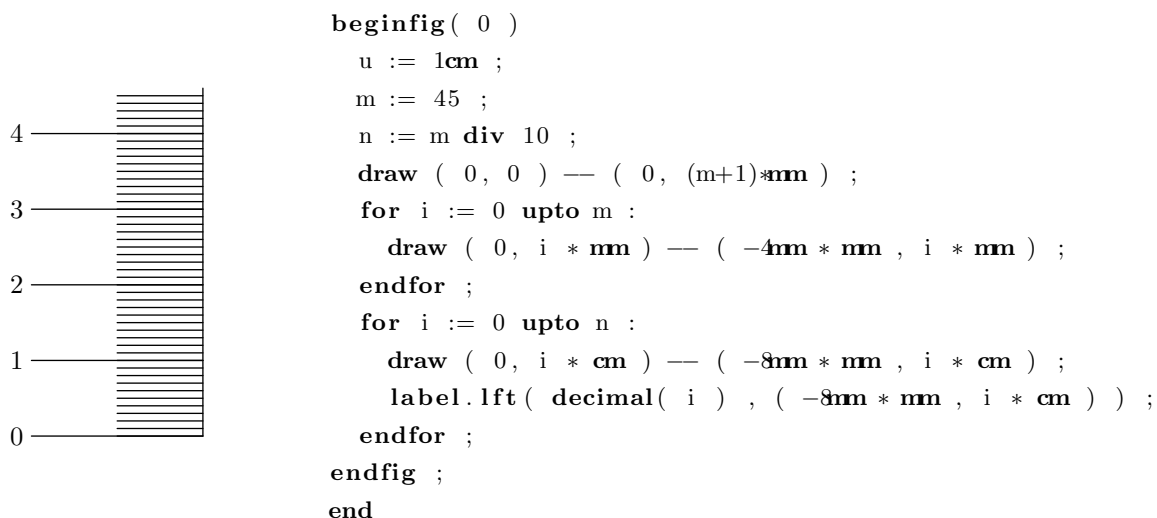
类似的，我们还有预先定义的 pensquare。通常我们把使用的画笔变粗了，会使得线条出现变化，转折的地方也会有所不同。

2.6 string 类型

string 类型就是用引号引起的，如 "hello"，虽然有时候并不能直接在引号里面写所有的 ASCII 码，但是可以用 **char**(n) 获得编码为 n 的 ASCII 码。同时可以利用 **decimal**(n) 将数字转换为对应的十进制字符串。图 15 就是用这个方法作出来的直尺刻度。

可以用 (m, n) **substring of** "string" 取出字符串的子串；**hex**("0xFE") 把一个十六进制字符串转换为 numeric 类型；**oct**("012") 把一个八进制字符串转换为 numeric 类型；**string**(var) 测试 var 是否为字符串。"string1" & "string2" 将会粘合两个字符串。

图 15: 一把直尺的刻度

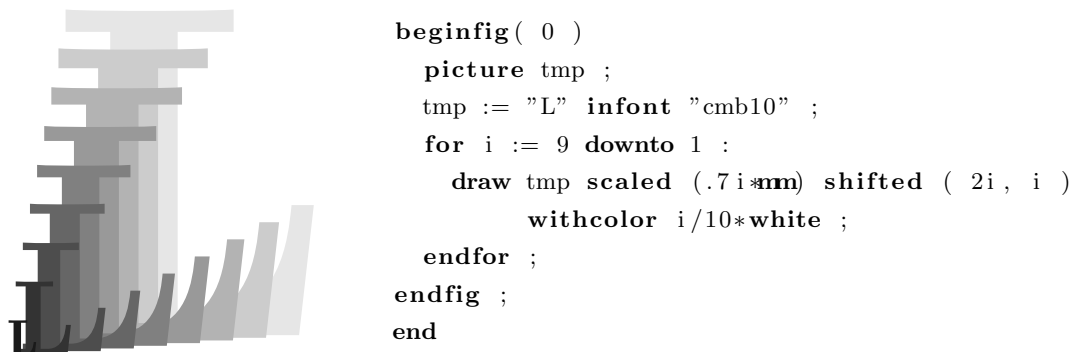


在 0.60 版之后的 METAPOST 中还可以通过 **readfrom** "file-name" 从文件中读字符串到 string 类型的变量, 每次读一行, 读到最后会产生一个空字符串。还可以利用 **write** "string" to "file-name" 把字符串写入到文件中, 每次写一行。

2.7 picture 类型

picture 类型作为一种类型, 可以被很多方式创建, 比如我们用命令画出来的整个东西就是一个 picture, 对应的变量名字为 **currentpicture**, 又比如我们使用 **btex etex** 包围的 T_EX/LaTeX 源码也可以产生一个 picture, 还可以利用字符串加对应的字体名, 例如 "text_here" **infont** "cmr9" 就是用 cmr9 创建前面字符串对应的图形了。图 16 中的例子为我们展示了如何利用 **infont** 产生的图片作图。这个例子需要查看第 5.1 小节的内容, 确保编译结果能为你所看见。

图 16: infont 创建的字体



另外一种构造 picture 的方式是利用 **addto** 命令将元素一个一个的加入到一个图形变量中。如 **addto picvar also pic**; 会将 **pic** 这个 picture 画入到 **picvar** 变量中。而 **addto picvar contour p** 则相当于在 **picvar** 上 **fill p** 的结果 (后面可以加其他的参数)。这些用法形成了 [Hobc] 中的表我们列在此处以供参考。

宏	相近命令
draw pic	addto picvar also pic
draw pa	addto picvar doublepath pa withpen q
fill cp	addto picvar contour cp
filldraw cp	addto picvar contour cp withpen q

表 1: **addto** 的等价用法

2.8 transform 类型

这里所说的 transform 正是二维平面上的仿射变换, 我们知道这种仿射变换需要六个变量确定:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} = \begin{pmatrix} ax + by + e \\ cx + dy + f \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix} \quad (1)$$

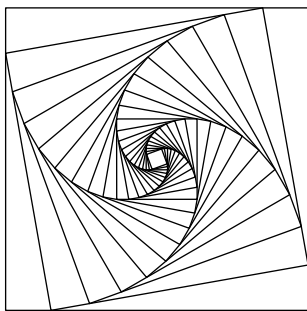
(1) 中的仿射变换可以用 METAPOST 的 (e, f, a, b, c, d) 表示。这个变换不仅仅可以作用在 pair 上, 还可以作用在 path、picture、pen 这些变量上产生对应的效果, 语法为

a transformed T

其中 a 为 pair、path、picture、pen 中任意一个, T 为 transform。类似与 pair 的 **xpart**, transform 的各个部分依次可以用 **xpart**, **ypart**, **xxpart**, **xypart**, **yxpart**, **yypart** 取出。

根据线性代数知识, 任意一个平面上的仿射变换可以由不共线的三点的像唯一确定, 证明可以由一个六元一次方程的解的唯一性得来。因此, 列出不共线三点在该变换下的像获得的三个方程, METAPOST 就可以确定该仿射变换了。图 17 的例子就说明了这种方式的可行性。

图 17: 不共线三点确定 transform



```

beginfig( 0 )
  u := 2cm ;
  path p ;
  transform t ;
  p := ( u, u ) — ( -u, u ) —
        ( -u, -u ) — ( u, -u ) — cycle ;
  ( 0, 0 ) transformed t = ( 0, 0 ) ;
  ( u, u ) transformed t = ( 0.7u, u ) ;
  ( -u, u ) transformed t = ( -u, 0.7u ) ;
  draw p ;
  for i := 1 upto 20 :
    p := p transformed t ;
    draw p ;
  endfor ;
endfig ;
end

```

另外, 为了更加方便的表达某些特定的 transform, 我们时常使用一些预先定义好的宏, 如 **scaled n** 表示线性放大比例为 n 倍, 又如 **shifted a** 表示平移到 a 处。其他的宏我们会在后面详细介绍。

值得注意的是，根据线性代数知识（看 (1)），该变换由一个线性变换（矩阵）加一个平移获得，而任意一个线性变换通过极分解成为一个旋转变换（正交变换）和一个伸缩变换（对应的是对称矩阵）复合而成，因此我们通常也可以用这些变换组合出任意的仿射变换。

2.9 数组

其实 METAPOST 的数组很简单，就是声明变量的时候后面加一个中括号，如 `pair a[]`；就声明了一个 `pair` 类型的数组，你可以用 `a[ind]` 来赋值、访问。所有的类型都可以产生数组。但是为了简便，甚至可以用 `a1` 表示 `a[1]`。但是和 C 的数组不大一样的是，这里的数组更像一个 C++ STL 的 `map<float, type>`，也就是说下标可以是任意数，你可以写 `a1.5`。

多维数组也可以类似的创建，如 `numeric n[][]`；甚至 `picture p[]q[]`；。

我们将用 Cantor 集（见图 18）这个例子说明如何利用数组作出分形。


图 18: Cantor 集合


```

beginfig( 0 )
  u := 5cm ;
  numeric p[] , q[] ;
  pen mypen ;
  p1 = 0 ; p2 := u ;
  m := 1 ;
  mypen := makepen( (0, 0) — (4mm, 0) — cycle ) ;
  for i := 1 upto 5 :
    for j := 1 upto m :
      q[4j-3] := p[2j-1] ;
      q[4j-2] := 1/3[ p[2j-1], p[2j] ] ;
      q[4j-1] := 2/3[ p[2j-1], p[2j] ] ;
      q[4j] := p[2j] ;
    endfor ;
    m := m * 2 ;
    for j := 1 upto 2m :
      p[j] := q[j] ;
    endfor ;
  endfor ;
  for i:= 1 upto m :
    draw ( 0, p[2i-1] ) — ( 0, p[2i] )
      withpen mypen ;
  endfor ;
endfig ;
end

```

Cantor 集是把一条单位长度的线段 $[0, 1]$ 中间的 $\frac{1}{3}$ 挖去，这样获得了两条线段： $[0, \frac{1}{3}]$ 、 $[\frac{2}{3}, 1]$ 。然后把这轮循环中获得的两条线段的中间的 $\frac{1}{3}$ 挖去，这将获得下一轮循环的若干条线段，每轮都把所得线段的中间的 $\frac{1}{3}$ 挖去。这样无止尽的挖下去就获得了 Cantor 集合。

 证明 Cantor 集剩余“线段”长度为 0，但是 Cantor 集中的点和原来 $[0, 1]$ 上的点一样多。该分形的 Hausdorff 维数为 $\frac{\log 2}{\log 3}$ 。

 看看你能不能画出本部分第一页上最下面的那条曲线，它叫 Koch 曲线，可以看做把一条单位长度的线段中间 $\frac{1}{3}$ 截去后添上两条相同长度的线段，这样在每轮获得的新线段上不断的进行该过程，进行无穷次迭代后获得的图形即为 Koch 曲线。该分形的维数为 $\frac{\log 4}{\log 3}$ 。

但是你要记住 METAPOST 的数组只是像数组，在这个宏语言中，更重要的是 suffix（后缀）概念，学习了后缀后你会发现，所谓的数组只是 suffix 的一个特例。

3 控制结构

3.1 条件控制

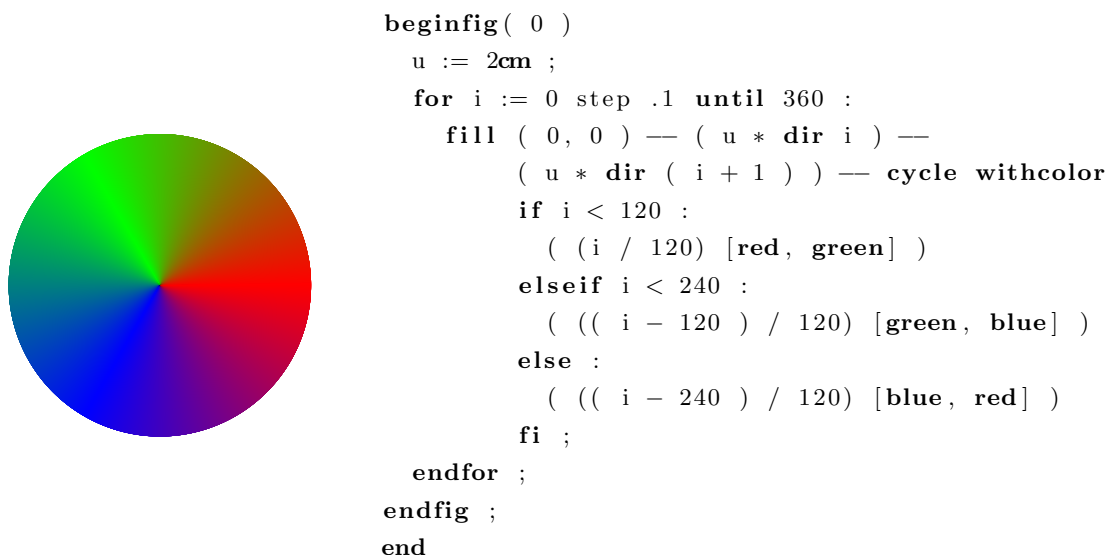
其实 METAPOST 可以使用 boolean 值, 如 `true`, `false`。因此, 你可以直接使用它们。另外, 比较运算 `<`, `=`, `>`, `<=`, `>=`, `<>`, 还有逻辑运算 `and`, `or`, `not`, 这样一来就可以利用这些逻辑运算的结果, 以及条件控制结构就可以形成各种分支。

最常见的条件控制结构就是用 `if` 引起的

```
if bool: ... fi
if bool: ... else: ... fi
if bool: ... elseif bool: ... fi
if bool: ... elseif bool: ... else: ... fi
```

但是 METAPOST 没有 C/C++ 中的 `switch-case` 控制结构。注意, 这种控制结构可以插到语句之中, 就好比 C/C++ 的那个 `?:` 算符作用一样, 图 19 就是用这种方式选择的不同颜色。

图 19: `if` 示意

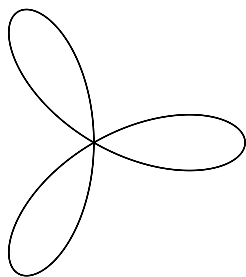


3.2 循环控制

这里主要有 `for`、`forever` 两种循环结构。对于 `for` 结构, 其实前面的例子中我们已经见到了很多用法, 最基本的用法就是 `for var := init step stepval until stop : ... endfor`, 该结构的一些简化写法是 `for var := init upto stopv : ... endfor` 和 `for var := init downto stopv : ... endfor`, 其中的 `upto` 也就是 `step 1 until`; `downto` 亦即 `step -1 until` 的写法。图 20 中就是一个将 `for` 插入到语句中的例子。其中的 `hide(expr)` 宏的作用在于不返回任何东西, 仅仅执行里面的操作, 因为这是在一个构造 `path` 变量的过程中, 只需要 `pair` 和 `--`。

其实, 这种 `for` 结构还可以写为更广义的 `for loopvar := valuelist : ... endfor`。其意义在于让 `loopvar` 遍历用, 分割的 `valuelist` 所有值, 并且类型没有任何要求, 也就是说可以遍历不同类型的常量、变量。

图 20: 三叶玫瑰线



```

beginfig( 0 )
  u := 2cm ;
  path p ;
  p := for i := 0 step 0.1 until 360 :
    hide( r:= u*cosd( 3i ); x:= r*cosd( i ) ;
          y:= r*sind( i ) ) ( x, y ) —
  endfor cycle ;
  draw p ;
endfig ;
end

```

如果想使用 **forever: endfor** 结构, 那么必须有一个退出循环体的方法, 那就是在循环的合适位置加入 **exitif** bool ; 或者 **exitunless** bool ; 。

另外还有一种 **forsuffixes** 的循环, 我们将在讲述了 **suffix** 概念后介绍。

3.3 自定义宏

3.3.1 def

基本用法是 **def** macroname = ... **enddef**。注意由于 METAPOST 是宏语言, 所以调用使用 **def** 定义的宏, 等价于把宏体写在调用处。因此, 表 1 中的命令都可以利用类似下面的方法定义

```
def fill = addto currentpicture contour enddef ;
```

我们也可以传递参数给 **def** 定义的宏, 如

```

def myshifted( expr p ) =
  transformed ( xpart p, ypart p, 0, 0, 0, 0 )
enddef

```

就可以获得一个类似于 **shifted** 的宏, 但是注意我们并没有检查传入参数 **p** 的类型。

正因为这是宏而不是函数, 所以在宏体里面改变了某个变量的值, 就会影响后面的值。



请利用 **show** 命令写一段简单的源文件证实这句话。

为了引入“函数”, 我们首先引入 **grouping** 的概念。

3.3.2 grouping

将几个基本语句组合起来, 并可能返回最后一句的值如果最后一句是一个表达式, 这就是一个 **group**。我们通过 **begingroup ... endgroup** 实现 **grouping**。图 21 说明了 **grouping** 的效果。在 **group** 里面, **show** x ; 显示的是一个字符串, 可见 **save** x ; 的作用在于清除掉了 **x** 的原始定义, 但是 **y** 被保留下来了, 从而在后面的改变 **y** 的值那句话的效果被延续到 **grouping** 完结之后。而 **x** 的值不受 **group** 内的影响, 在 **group** 里面新定义的变量却会保留到 **grouping** 完结之后, 如 **w**。

在 [Hobc] 中提到, **whatever** 可以用这种策略产生

```

begingroup
  save ? ; ?
endgroup

```

图 21: 一个 grouping 示例

```

( grouping .mp
>> x
>> 2
>> 1
>> 3
>> 5 )

x := 1 ;
y := 2 ;
begingroup
  save x ;
  show x ;
  show y ;
  y := y + 1 ;
  x := 4 ;
  w := 5 ;
endgroup ;
show x ;
show y ;
show w ;
end

```

但是 METAPOST 的内部变量，如前面提到的 `tracingonline` 不可以用 `save`，必须用 `interim`。这在图 22 可以比较的很清楚。内部变量 `linecap` 控制着线条端点的形状，使用 `linecap := butt` 时，画出的最下面一根线是平头，再看进入了第一个 group 中，使用 `save linecap`；重定义了 `linecap := rounded`；，虽然 `show linecap`；显示的结果已经不同了，但是画出来的并不是我们需要的圆头，这说明定义的内部变量没有改变内部变量的值，只是“屏蔽”了我们对它的访问。再看第二个 group，使用了 `interim linecap:= rounded`；结果就对了。最后画的那条线段表明，`linecap` 只是临时变化了。

完成了 grouping，在我们定义一个需要用到局部变量时，就知道会去用这种策略了。

3.3.3 expr、text 和 suffix

前面没有解释 `expr` 什么意思，这里将进行详细的解释。其实传递给宏的参数一共有三种类型，`expr`、`text`、`suffix`。

`expr` 说穿了就是一个表达式的值，比如你有 `x := 1`；，用 `expr` 类型传 `x` 进去，结果就是传了它的值 1 进去，这和 C++ 中一般传值是一致的。`text` 类型的参数就是不管什么都可以传，原来是什么就放进去是什么，变量也好，表达式也好，所以这个类型变量一用，后面的肯定匹配不了其他的变量了，一般仅仅单用。`suffix` 类型的参数比较明晰，类似于 C++ 中传引用类型，因此只适合于变量。

关于 `text` 类型传参数，最明显的例子就是 [Hobc] 和 [Hec] 中都提到的，在图 20 中也使用到的 `hide()` 宏，这里给出 [Hec] 中的定义方式（在我的 tetex 3.0 的 `plain.mf` 文件中也是如此定义的）。

```
def hide(text t) = exitif numeric begingroup t; endgroup; enddef;
```

又如

```
def drawall( text t ) =
  for i := t :
    draw i ;
  endfor ;
enddef ;
drawall( ( 0, 0 ) — ( 10, 0 ), btex $x^2$ etex,
  "this_is_a_joke" infont "cmr10" ) ;
```

图 22: save 与 interim

```

(save-interim.mp
>> 0
>> 1
>> 1
>> 0 [0] )

beginfig( 0 )
  u := 1cm ;
  pickup pencircle scaled 5mm ;
  linecap := butt ;
  draw ( 0, 0 ) — ( u, 0 ) ;
  show linecap ;
  begingroup
    save linecap ;
    linecap := rounded ;
    draw ( 0, u ) — ( u, u ) ;
    show linecap ;
  endgroup ;
  begingroup
    interim linecap := rounded ;
    draw ( 0, 2u ) — ( u, 2u ) ;
    show linecap ;
  endgroup ;
  draw ( 0, 3u ) — ( u, 3u ) ;
  show linecap ;
endfig ;

end

```

为了比较 `expr` 和 `suffix` 可以执行下面的程序看看错误怎么解决。

```

def f( expr a ) = a := 2 ; enddef ;
def g( suffix a ) = a := 3 ; enddef ;
x := 1 ; show x ;
f( x ) ; show x ;
g( x ) ; show x ;
end

```

为了能用上两种类型的参数，你需要用两个括号分别声明

```
def mymacro( expr t )( suffix s ) = ... enddef ;
```

调用的时候写成 `mymacro(t, s)` 和 `mymacro(t)(s)` 是一样的。

3.3.4 vardef

当然，可以简单的认为 `vardef macroname(...) = ... enddef`；是和

```

def macroname( ... ) =
  begingroup
    ...
  endgroup ;
enddef ;

```

产生的效果类似，但是实际最大的不同在于 `vardef` 定义的宏是类似于 C++ 中的 `template` 的，也就是说这个宏是可以加“参数”变换成新的宏，或者说是一个宏的模板。我们援引 [Hobc] 中的例子说

明这一点

```
vardef a[]b( expr p ) = p shifted( #@, b ) enddef ;
```

注意到 `a[]b` 中的中括号，那就是一个可以加入“模板参数”的地方，如你可以使用 `a3b(c)`, `a.4b(c)`，不信？试试这个。怎么？开始觉得奇怪了？嗯，`#@` 是什么意思呢？其实它代表的是最后一个 `[]` 之前的（包括 `[]`）所有东西，这里是 `a3` 和 `a.3`，而另外有一个 `@` 表示最后 `[]` 之后的所有东西了，这里就是 `b` 了。

但是如果 `[]` 在最后，那么 `#@` 是 `[]` 之前的东西（不包括 `[]`），而 `@` 就是 `[]` 了。比如前面的例子写成 `vardef a.b[] = ... enddef` 时，调用 `a.b2` 时，`#@` 是 `a.b`，而 `@` 为 `[2]`，为什么不是 `2` 呢？注意，不知道你有没有发现这么一个规律，METAPOST 的 suffix（后缀）如果是字母，可以直接连在前面的写，如数组（注意，这里的数组是用宏模拟出来的），你会写 `a[1]`, `a1`, `b1 2`。如果 suffix 是字母，就得用 `a.bot` 等方式来写了，要把这个后缀转换为我们能处理的东西，我们少不了用 `str @`，怎么？是不是又觉得有些不大好理解，试试这个。你会发现这样定义的不能加 `a.b.c` 的后缀。但是 numeric 都可以。

为了得到上面不成功类型的后缀，我们可以用 `@#` 作为定义的后缀，如

```

vardef house@# ( expr p ) =
  p shifted ( if str@# = "window" : ( 2, 0 ) + fi ( 1, 3 ) )
enddef ;
>> ( 1,3) show house( origin ) ;
>> ( 3,3) show house.window( origin ) ;
>> ( 1,3 ) show house.widow( origin ) ;
end
```

METAPOST 使用 `@#` 定义的宏里面有一个是

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

注意，`z0` 不是一个变量，而是一个宏！不信试试这个吧。

这里我们看看 `forsuffixes` 的用法，从中我们会更好的理解 `@#` 的用法，下面是一个为多个点同时标注的宏

```
vardef dotlabels@#(text t) =
  forsuffixes $=t:
    dotlabel@#(str$, z$); endfor
enddef;
```

这里将前面所讲到的知识基本全用上了，首先传入的是 `text` 型参数，因此，你可以随意写传多个值进入。然后 `forsuffixes` 循环中的变量 `$` 把 `t` 中每个转换为 suffix，而 `dotlabels` 的后缀将变为 `dotlabel` 的后缀，其参数有两个，一个是用 `str$` 转换后缀为字符串，一个是用 `z$` 转换为需要的 pair。`dotlabel` 的意思就是在第二个参数附近写一个标注，从而 `dotlabels` 的意思就是在 `z` 系列 `((x$,y$))` 点附近写上标注。如 `dotlabels(1, 2, b)`，就会在 `z1`, `z2`, `z.b` 附近标注。

如果你认为这部分的作图实例少了，请参考第 7 节。

3.3.5 算符

算符和宏的一点点区别在于调用方法，如我们常称 `round a` 是一个算符，而 `round(a)` 为宏，其实在 METAPOST 里面两者定义方式差得不远，首先我们看看定义的时候会不会有差别，我们使用 `def` 来定义一个返回一个比参数大 1 的数。然后使用不同的调用方式，

```

tracingall ; showstopping := 0 ;
def f(expr t) = t + 1 enddef ;
def g expr t = t + 1 enddef ;
x := 1 ;
show f( x ) ; show f x ;
show g( x ) ; show g x ;
end

```

你会发现使用 `def f()` 定义的宏不能充当算符，因为调用 `f x` 出错。但是 `g x` 和 `g(x)` 都是可以的。

那么，你会不会以为这就是我们需要的算符了呢？我们继续一个实验，见图 23，为什么两者显示的结果不一样呢？注意到声明中一个是 `expr` 而另一个是 `primary` 了么？

图 23: `primary` vs `expr`

<pre> (expr-primary.mp >> 10 >> 7) </pre>	<pre> def f expr t = t * 2 enddef ; def g primary t = t * 2 enddef ; x := 2 ; show f x + 3 ; show g x + 3 ; end </pre>
--	---

下面，让我们来解释一下 METAPOST 的解析顺序吧，首先我们看看从 [Hobc] 中截取出来的关系图（参见图 24），首先是 `atom`，然后是 `primary`、`secondary`、`tertiary`、`subexpression`，最后才是 `expression`，换言之，METAPOST 依照这样的顺序来解析你的源文件，你可以试试这个文件，这里面定义了四个算符，分别是 `primary`、`secondary`、`tertiary` 和 `expression (expr)`。然后你会发现我们使用了不同的运算和这四个算符结合，结果是很不一样的，优先顺序是前面的高，后面的低。我们先从图 23 看起，对 `f x + 3` 而言，`f` 是一个 `expression` 级别的算符，`+` 是 `secondary` 的，所以先进行 `x + 3` 成为了 5，最后结果是 10。而对 `g x + 3`，先算 `g x` 得 4 所以结果为 7。尝试过我们给的测试文件后，你会更加明白不同级别运算的区别。

对于二元算符，我们使用 `primarydef`、`secondarydef`、`tertiarydef`，注意，这里没有 `def` 或者 `vardef` 对应的版本了，比如我们可以简单的定义两个 `pair` 的点积

```
primarydef w dotprod z = xpart w * xpart z + ypart w * ypart z enddef ;
```

其中的 `z` 并不是宏。在 [Hobc] 给出了这些算符之间的优先顺序，我们将其摘录在图 25。

在图 25 中最后一项是所谓的 `of operator`，注意 `of` 是 METAPOST 保留字（你可以尝试对它赋值，然后 METAPOST 会报错），这里我们看一个简单的 `of operator` 的定义，

```
def sq expr t of p = p ** t enddef ;
```

这时，我们可以用 `sq 2 of 3` 来计算 3^2 了，那么 `of` 的前后是什么呢？打开了调试功能（参考第 6 节），你就会看到一个类似的执行结果，

```

sq<expr>of<primary>->(EXPR1)**(EXPR0)
(EXPR0)<-2
(EXPR1)<-3

```

可见 `of` 前后分别是 `expression` 和 `primary` 类型的。

METAPOST 中的宏、算符都是可以通过重定义消除原先的含义，这也就为我们后面通过重定义某些系统的宏实现自定义的图形（如箭头）提供了保证。

$\langle \text{atom} \rangle \rightarrow \langle \text{variable} \rangle \mid \langle \text{argument} \rangle$
 $\mid \langle \text{number or fraction} \rangle$
 $\mid \langle \text{internal variable} \rangle$
 $\mid \langle \langle \text{expression} \rangle \rangle$
 $\mid \text{begingroup} \langle \text{statement list} \rangle \langle \text{expression} \rangle \text{endgroup}$
 $\mid \langle \text{nullary op} \rangle$
 $\mid \text{bte} \langle \text{typesetting commands} \rangle \text{etex}$
 $\mid \langle \text{pseudo function} \rangle$
 $\langle \text{primary} \rangle \rightarrow \langle \text{atom} \rangle$
 $\mid \langle \langle \text{numeric expression} \rangle, \langle \text{numeric expression} \rangle \rangle$
 $\mid \langle \langle \text{numeric expression} \rangle, \langle \text{numeric expression} \rangle, \langle \text{numeric expression} \rangle \rangle$
 $\mid \langle \text{of operator} \rangle \langle \text{expression} \rangle \text{of} \langle \text{primary} \rangle$
 $\mid \langle \text{unary op} \rangle \langle \text{primary} \rangle$
 $\mid \text{str} \langle \text{suffix} \rangle$
 $\mid \text{z} \langle \text{suffix} \rangle$
 $\mid \langle \text{numeric atom} \rangle [\langle \text{expression} \rangle, \langle \text{expression} \rangle]$
 $\mid \langle \text{scalar multiplication op} \rangle \langle \text{primary} \rangle$
 $\langle \text{secondary} \rangle \rightarrow \langle \text{primary} \rangle$
 $\mid \langle \text{secondary} \rangle \langle \text{primary binop} \rangle \langle \text{primary} \rangle$
 $\mid \langle \text{secondary} \rangle \langle \text{transformer} \rangle$
 $\langle \text{tertiary} \rangle \rightarrow \langle \text{secondary} \rangle$
 $\mid \langle \text{tertiary} \rangle \langle \text{secondary binop} \rangle \langle \text{secondary} \rangle$
 $\langle \text{subexpression} \rangle \rightarrow \langle \text{tertiary} \rangle$
 $\mid \langle \text{path expression} \rangle \langle \text{path join} \rangle \langle \text{path knot} \rangle$
 $\langle \text{expression} \rangle \rightarrow \langle \text{subexpression} \rangle$
 $\mid \langle \text{expression} \rangle \langle \text{tertiary binop} \rangle \langle \text{tertiary} \rangle$
 $\mid \langle \text{path subexpression} \rangle \langle \text{direction specifier} \rangle$
 $\mid \langle \text{path subexpression} \rangle \langle \text{path join} \rangle \text{cycle}$

图 24: METAPOST 中的词法

$\langle \text{nullary op} \rangle \rightarrow \text{false} \mid \text{normaldeviate} \mid \text{nullpicture} \mid \text{pencircle}$
 $\mid \text{true} \mid \text{whatever}$
 $\langle \text{unary op} \rangle \rightarrow \langle \text{type} \rangle$
 $\mid \text{abs} \mid \text{angle} \mid \text{arclength} \mid \text{ASCII} \mid \text{bbox} \mid \text{bluepart} \mid \text{bot} \mid \text{ceiling}$
 $\mid \text{center} \mid \text{char} \mid \text{cosd} \mid \text{cycle} \mid \text{decimal} \mid \text{dir} \mid \text{floor} \mid \text{fontsize}$
 $\mid \text{greenpart} \mid \text{hex} \mid \text{inverse} \mid \text{known} \mid \text{length} \mid \text{lft} \mid \text{llcorner}$
 $\mid \text{lrcorner} \mid \text{makepath} \mid \text{makepen} \mid \text{mexp} \mid \text{mlog} \mid \text{not} \mid \text{oct} \mid \text{odd}$
 $\mid \text{redpart} \mid \text{reverse} \mid \text{round} \mid \text{rt} \mid \text{sind} \mid \text{sqrt} \mid \text{top} \mid \text{ulcorner}$
 $\mid \text{uniformdeviate} \mid \text{unitvector} \mid \text{unknown} \mid \text{urcorner} \mid \text{xpart} \mid \text{xxpart}$
 $\mid \text{xy part} \mid \text{ypart} \mid \text{yxpart} \mid \text{yypart}$
 $\langle \text{type} \rangle \rightarrow \text{boolean} \mid \text{color} \mid \text{numeric} \mid \text{pair}$
 $\mid \text{path} \mid \text{pen} \mid \text{picture} \mid \text{string} \mid \text{transform}$
 $\langle \text{primary binop} \rangle \rightarrow * \mid / \mid ** \mid \text{and}$
 $\mid \text{dotprod} \mid \text{div} \mid \text{infon} \mid \text{mod}$
 $\langle \text{secondary binop} \rangle \rightarrow + \mid - \mid ++ \mid +- \mid \text{or}$
 $\mid \text{intersectionpoint} \mid \text{intersectiontimes}$
 $\langle \text{tertiary binop} \rangle \rightarrow \& \mid < \mid \leq \mid <> \mid = \mid > \mid \geq$
 $\mid \text{cutafter} \mid \text{cutbefore}$
 $\langle \text{of operator} \rangle \rightarrow \text{arctime} \mid \text{direction} \mid \text{directiontime} \mid \text{directionpoint}$
 $\mid \text{penoffset} \mid \text{point} \mid \text{postcontrol} \mid \text{precontrol} \mid \text{subpath}$
 $\mid \text{substring}$

图 25: 算符优先顺序

4 作图基本知识

前面我们在讲述 pen 类型的时候已经涉及到作图的问题了。作图好比是你拿着一只特定的笔，在一个画布上按照一定的规定写字，如转折的时候，书法中不同体例要求的转折方式不尽相同，最后封头的方式（如一横或者一竖的结尾）也不尽相同。还有使用的线型，如什么样的虚线，要不要箭头，这些构成了作图的基本问题。

4.1 选择什么样的笔？

如果你需要创作带艺术风格的图画，可能笔的样子会很重要，这里仅仅举几个例子说明，因为一般我们都不需要过分的调整笔的样式。

这是 [Hec] 提供的例子：

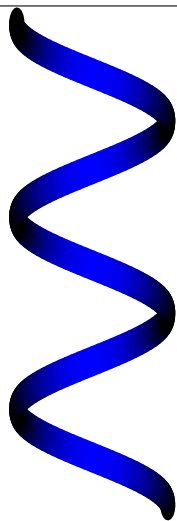


```

beginfig(1)
  pickup pencircle xscaled 4bp
    yscaled 0.5bp rotated 60 ;
  for i=10 downto 1:
    draw 5(i,0) .. 5(0,i) .. 5(-i,0)
      .. 5(0,-i+1) .. 5(i-1,0)
      withcolor red;
  endfor
endfig;
end

```

又比如结合合适的配色可以形成立体感（当然这个例子不是很好），



```

beginfig( 0 )
  u := 1cm ;
  for i := 0 step 1 until 180 :
    draw ( u * cosd( 5i ), i ) — ( u * cosd( 5i + 5 ), i+1 )
      withpen pencircle yscaled 4mm xscaled 2mm
      withcolor ( ( cosd( 5i )** 2 ) [blue, black] ) ;
  endfor ;
endfig ;
end

```

4.2 线型

所谓的线型本质上是一个 picture 类型的值，我们描述线型，本质上就是形成一个重复出现的 picture 供我们作图使用。最常见的命令就是 **dashpattern**，它为我们提供了一个一般性的产生不同线型的方法，其基本用法是

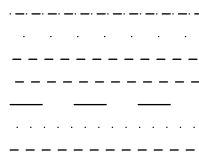
```
dashpattern( on-off-list )
```

其中的参数是描述什么时候画实线，什么时候不画线，如 `on 6bp off 12bp on 6bp` 就是画 6bp 长度的实线，然后停下来向前走 12bp，然后再画 6bp 的实线。这样的线型重复出现，我们就看见了 12bp 实线，空 12bp 的虚线了。这里我们可以看看系统两种常见线型是如何定义的

```
picture evenly, withdots;
evenly = dashpattern(on 3 off 3);
withdots = dashpattern(off 2.5 on 0 off 2.5);
```

图 26 向我们演示了不同线型作图的结果。

图 26: 不同线型作图



```
beginfig( 0 )
  u := 1cm ;
  def drawone( expr h, dp ) =
    draw ( 0, h ) — ( 2.5u, h ) dashed dp ;
  enddef ;
  drawone( 0u, evenly ) ;
  drawone( .3u, withdots ) ;
  drawone( .6u, evenly scaled 4 ) ;
  drawone( .9u, evenly shifted (2bp, 0 ) ) ;
  drawone( 1.2u, evenly shifted (1bp, 0 ) ) ;
  drawone( 1.5u, withdots scaled 2 ) ;
  drawone( 1.8u, dashpattern(
    on 0 off 2bp on 3bp off 1bp ) ) ;
endfig ;
end
```

为了更好的理解 `dashpattern`，我们看看其定义方式，

```
vardef dashpattern(text t) =
  save on, off, w;
  let on=_on_;
  let off=_off_;
  w = 0;
  nullpicture t
enddef;

tertiarydef p _on_ d =
  begingroup save pic;
  picture pic; pic=p;
  addto pic doublepath (w,w)..(w+d,w);
  w := w+d;
  pic shifted (0,d)
endgroup
enddef;

tertiarydef p _off_ d =
  begingroup w:=w+d;
  p shifted (0,d)
endgroup
```

enddef;

可见, `dashpattern` 返回的是一个 `picture` 类型, 那么这个 `picture` 类型怎么被系统使用呢? 我们拿 `evenly` 来说明。首先我们计算它产生的 `picture` 是什么样子的: 根据上面 `evenly` 的定义, 相当于调用了 `nullpicture _on_ 3 _off_ 3`, 计算两个算符, `nullpicture _on_ 3` 就是先画了一个直线 $(0, 0) \text{--} (3, 0)$, 然后 `w:=3`; 返回的图片将原来的线段变成了 $(0, 3) \text{--} (3, 3)$; 再算第二个算符, `w:=6`, 再把图片平移, 那根线段到了 $(0, 6) \text{--} (3, 6)$ 。图 27 展示了这个事实。其中, `urcorner` 返回一个 `picture` 的右上

图 27: 线型研究

<pre>(dashpattern.mp >> (3,6) >> (0,6) >> (9,9) >> (0,9))</pre>	<pre>picture p ; show urcorner evenly ; show llcorner evenly ; p := dashpattern(on 3 off 3 on 3) ; show urcorner p ; show llcorner p ; end</pre>
--	--

角坐标, `llcorner` 返回左下角坐标。那么 METAPOST 怎么利用这个 `picture` 作图呢? 首先利用 y 轴坐标确定这个线型的长度, 然后把线段投影到 x 轴就获得了重复的线型。我们可以做两个实验验证我们的想法, 第一个已经很明显, y 轴坐标是线型总长度的来源, 那么是哪一点的 y 轴坐标呢? 图 28 为我们说明了这个问题, 其中的黑点就是原点了。看来选择的就是这个 `picture` 的 `llcorner` 的 y 坐标。第二个, 同样从图上看出。

图 28: dashed 怎么利用线型

(a) (b)

```
beginfig( 0 )
picture mypattern ;
mypattern := nullpicture ;
addto mypattern doublepath
( 0, 5mm ) -- ( 5mm, 10mm ) ;
draw mypattern ;
draw ( 0, 1cm ) -- ( 0, 3cm )
dashed mypattern ;
fill fullcircle scaled 1mm ;
endfig ;
beginfig( 1 )
picture mypattern ;
mypattern := nullpicture ;
addto mypattern doublepath
( 2.5mm, 5mm ) -- ( 5mm, 10mm ) ;
draw mypattern ;
draw ( 0, 1cm ) -- ( 0, 3cm )
dashed mypattern ;
fill fullcircle scaled 1mm ;
endfig ;
end
```

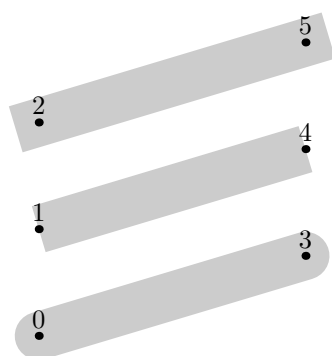
不能使用太复杂的 `picture` 作为 `dashed` 的参数, 否则它会抱怨

! Picture is too complicated to use as a dash pattern.

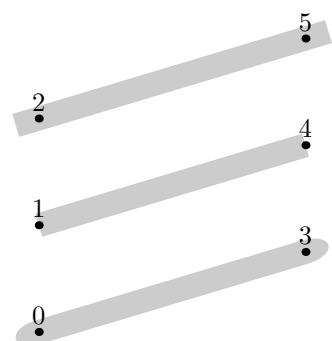
4.3 端点

早在图 22 中我们就发现了 `linecap` 这个内部变量控制着线段的端点的形状。图 29 中这个来自 [Hobc] 的例子说明了三种不同类型的端点样式。

图 29: 端点样式



(a)



(b)

```
beginfig(0)
  for i=0 upto 2:
    z[i] = ( 0, 40i );
    z[i+3] - z[i] = ( 100, 30 );
  endfor ;
  pickup pencircle scaled 18 ;
  draw z0 .. z3 withcolor .8white ;
  linecap := butt ;
  draw z1 .. z4 withcolor .8white ;
  linecap:=squared ;
  draw z2 .. z5 withcolor .8white ;
  dotlabels.top( 0, 1, 2, 3, 4, 5 ) ;
endfig;

beginfig(1)
  linecap := rounded ;
  for i=0 upto 2:
    z[i] = ( 0, 40i );
    z[i+3] - z[i] = ( 100, 30 );
  endfor ;
  pickup pencircle xscaled 18
    yscaled 9 rotated angle(100, 30) ;
  draw z0 .. z3 withcolor .8white ;
  linecap := butt ;
  draw z1 .. z4 withcolor .8white ;
  linecap:=squared ;
  draw z2 .. z5 withcolor .8white ;
  dotlabels.top( 0, 1, 2, 3, 4, 5 ) ;
endfig;

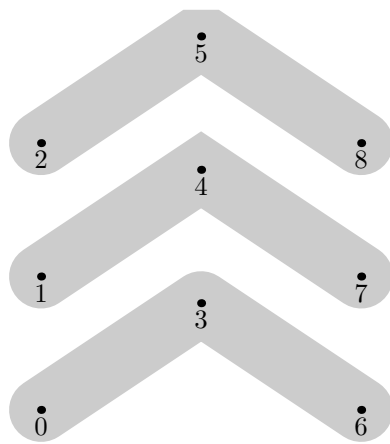
end
```

这个样式和使用的画笔形状有关吗？这个问题在图 29 中后面的例子就能看出来，`butt` 将多出部分完全切掉，因此不受影响，而 `rounded` 属于完全不管闲，原来是什么样就是什么样，而 `squared` 却是延伸出去和 `rounded` 一般长，但是为方块头。

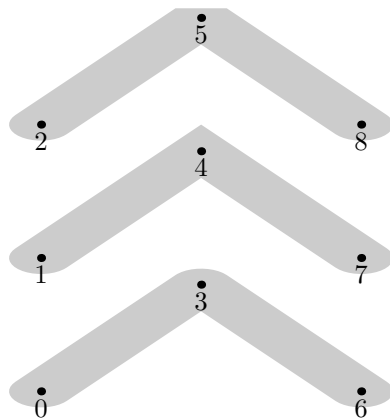
4.4 接头

接头就是在线段转弯的时候使用的连接形式。我们仍然用 [Hobc] 中的例子，见图 30。

图 30: 接头样式



(a)



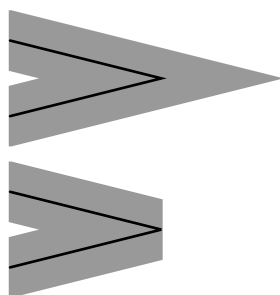
(b)

```

beginfig( 0 )
  for i := 0 upto 2 :
    z[i]=( 0, 50i ) ;
    z[i+3] - z[i] = ( 60, 40 ) ;
    z[i+6] - z[i] = ( 120, 0 ) ;
  endfor ;
  pickup pencircle scaled 24;
  draw z0--z3--z6 withcolor .8white;
  linejoin:=mitered;
  draw z1..z4--z7 withcolor .8white;
  linejoin:=beveled;
  draw z2..z5--z8 withcolor .8white;
  dotlabels.bot(0,1,2,3,4,5,6,7,8);
endfig ;
linejoin:=rounded;
beginfig( 1 )
  for i := 0 upto 2 :
    z[i]=( 0, 50i ) ;
    z[i+3] - z[i] = ( 60, 40 ) ;
    z[i+6] - z[i] = ( 120, 0 ) ;
  endfor ;
  pickup pencircle xscaled 24 yscaled 12 ;
  draw z0--z3--z6 withcolor .8white;
  linejoin:=mitered;
  draw z1..z4--z7 withcolor .8white;
  linejoin:=beveled;
  draw z2..z5--z8 withcolor .8white;
  dotlabels.bot(0,1,2,3,4,5,6,7,8);
endfig ;
end

```

可以看出 **rounded** 基本就是用原来 pen 的形状放在角平分线的位置, 平滑的结果。而 **beveled** 就是沿外角平分线切下, **mitered** 是延长直到相交, 但是我们知道这个转角越接近 180° 就越会产生很长的尖角越长, 这个长度有个限制, 由 **miterlimit** 控制。这个从图 31 可以看出。该变量是管理尖角两个尖尖之间距离与线宽的比例, 当该比例不超过这个值的时候, 可以形成尖角; 否则则用 **beveled** 的方式处理。

图 31: **miterlimit** 的影响

```
beginfig( 0 )
  path p ;
  linejoin := mitered ;
  miterlimit := 3 ;
  p := ( 0, -5mm ) -- ( 2cm, 0 ) -- ( 0, 5mm ) ;
  pickup pensquare yscaled 8mm ;
  draw p withcolor 0.6white ;
  pickup pencircle ;
  draw p ;
  pickup pensquare yscaled 8mm ;
  miterlimit := 10 ;
  draw p shifted ( 0, 2cm ) withcolor 0.6white ;
  pickup pencircle ;
  draw p shifted ( 0, 2cm ) ;
endfig ;
end
```

4.5 箭头

把 **draw** 换成 **drawarrow** 或者 **drawdblarrow**, 画出来的路径的末尾或者两头就会出现箭头。这两个命令的定义如下:

```
path _apth;
def drawarrow expr p = _apth:=p; _finarr enddef;
def drawdblarrow expr p = _apth:=p; _findarr enddef;

def _finarr text t =
  draw _apth t;
  filldraw arrowhead _apth t
enddef;

def _findarr text t =
  draw _apth t;
  filldraw arrowhead _apth withpen currentpen t;
  filldraw arrowhead reverse _apth withpen currentpen t
enddef;
```

由此我们可以看出, 控制箭头形状的在 **arrowhead** 这个 operator 里面, 这是 **arrowhead** 的定义

```
vardef arrowhead expr p =
  save q,e; path q; pair e;
  e = point length p of p;
```



```

q = gobble(p shifted -e cutafter makepath(pencircle scaled 2ahlength))
  cuttings;
(q rotated .5ahangle & reverse q rotated -.5ahangle — cycle) shifted e
enddef;

```

我们来分析一下这段代码的作用，由于出现了很多其他的宏，具体是什么请参考 7 一节，这里简要说明一下，首先 `e` 是输入路径的终点，因为 `length p` 返回 `p` 路径的参数范围最大值，所以 `point of` 这个 `of operator` 取出了 `p` 的终点。`gobble` 宏就是什么都不干，它可能比 `hide` 还要懒，那么它有什么用呢？注意到 `gobble` 里面的 `expression` 这时会执行，最主要的是那个 `cutafter` 会执行，其结果就是被剪下来的一段路径，存放在一个系统定义好的 `path` 变量 `cuttings` 中，所以 `q` 是被剪下来的一段路径，那么什么被剪下来了呢？`p shifted -e` 将 `p` 平移使得终点到达原点，这就是被剪的对象（`cutafter` 之前）；用什么剪？自然是后面的 `makepen(pencircle scaled 2ahlength)` 这个圆心在原点，半径是预先指定的 `ahlength`（表示箭头大小）的圆。`cutafter` 的意思就是把这个圆后面的剪掉，所以，圆内的部分（就是 `p` 末尾的一小节）成为了 `q`。最后返回的是一个闭合的路径，它由三条边组成，两条是把 `q` 绕 `e` 向两侧转动 `ahangle` 的一半大小，另外一条就是用条直线连接这两条线剩余的一端。

好了，这下你就能理解 [Hobc] 中间这幅图片了，参见图 32。其中那些不熟悉的宏，请参考第 7 节。

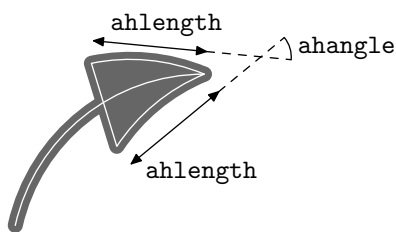
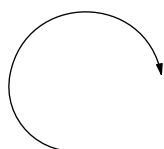


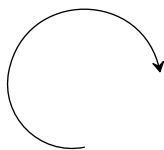
图 32: 箭头示意

看明白了别人的箭头怎么定义的，你会不会也想画一个自己的箭头玩玩呢？图 33 中的例子是参考 [Zoo] 里面的若干例子实现一些自己的箭头。

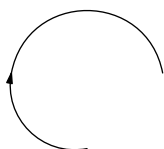
图 33: 自定义箭头



(a)



(b)



(c)

```

beginfig( 0 )
  save arrowhead ;
  vardef arrowhead expr p =
    save A, u ;
    pair A, u ;
    A := point length( p ) of p ;
    u := unitvector( direction length( p ) of p ) ;
    A — (A — ahlength*u rotated 15) —
      ( A — ahlength*u rotated -15 ) — cycle
  enddef;
  u := 1cm ;
  drawarrow ( 0, 0 ) .. ( -u, u ) .. ( u, u ) ;
endfig;

beginfig( 1 )
  save arrowhead ;
  vardef arrowhead expr p =
    save A, B, u ;
    pair A, B, u ;
    A := point length(p) of p ;
    B := p intersectionpoint
      ( fullcircle scaled ahlength shifted A ) ;
    u := unitvector(direction length( p ) of p ) ;
    A — (A — ahlength*u rotated 30) — B —
      ( A — ahlength*u rotated -30 ) — cycle
  enddef;
  u := 1cm ;
  drawarrow ( 0, 0 ) .. ( -u, u ) .. ( u, u ) ;
endfig;

beginfig( 2 )
  save arrowhead ;
  vardef arrowhead expr p =
    save A, u ;
    pair A, u ;
    A := point 1/2length( p ) of p ;
    u := unitvector( direction 1/2length(p) of p ) ;
    A — (A — ahlength*u rotated 15) —
      (A — ahlength*u rotated -15) — cycle
  enddef;
  u := 1cm ;
  drawarrow ( 0, 0 ) .. ( -u, u ) .. ( u, u ) ;
endfig;

end

```

第二部分 实践篇

5 FAQ

这里搜罗一些常见的使用 METAPOST 过程中常见的问题。

5.1 字体显示不对或者不能显示

在使用 METAPOST 的 `btex ... etex` 结果时，有时候会碰到显示的字体并非所需要的或者根本不能显示，那么先看看你的文件中有没有

```
prologues := 1 ;
```

如果没有，请加上。

如果显示不正常，说明 GSView 使用的（其实 ImageMagick/GraphicsMagick 也是用的）ghostscript 没能找到相关字体，因此，需要做一些配置。按照 王垠的主页 上的方式，一可以直接用

```
export GSFONTPATH=$TEXMF/fonts/type1/bluesky/cm
```

加入自己的路径，不同的使用冒号分割。

但是这种方式会影响 GSView 的启动速度，所以一般建议使用下面的办法。在每一个装有字体的目录中，如上面的 `$TEXMF/fonts/type1/bluesky/cm` 加入一个 Fontmap 文件。其格式可以参考如下脚本产生的结果：

```
#!/bin/sh

for i in $(ls *.pfb)
do
    fn=$(basename $i .pfb)
    FN=$(echo $fn | tr [:lower:] [:upper:] '
    echo "$FN_(${fn}.pfb)";"
done
```

使用的时候，可以使用

```
$ genFontList.sh /usr/share/texmf/fonts/type1/bluesky/cm > Fontmap
$ sudo cp Fontmap /usr/share/texmf/fonts/type1/bluesky/cm
$ rm Fontmap
```

5.2 合适的 editor

5.3 如何在 PDF_LA_TE_X 中使用 METAPOST?

6 调试

7 常用宏

8 常见宏包简介

第三部分 综合篇

9 使用 *METAPOST* 制作精美动画

10 Gallery

11 C++ flavor: Asymptote

第四部分 附录

A 贡献者

这里要感谢 aloft 老大为大家提供的 \TeX 论坛，以及论坛上面一些热心的网友，这里列出的仅仅是我记得的给予本文直接或者间接帮助的人们。

- Neals, 首先对本文的排版提出了一些意见，如行文中的关键字最好用等宽字体，和我讨论了一些关于加入 `label` 后编译的方案。
- changroc, 提供了一个法文的 METAPOST 网站。
- elove, 是 m3D 和 mol3D 的最初介绍者，也是 \TeX 论坛上 METAPOST 版面的主要支撑者。
- yg, 提供了大量的 Asymptote 示例和源码，使我不必四处搜索各方面的资料。

感谢大家对本文的支持，写作的很大一部分动力来源于你们的需要和称赞。

最后感谢偶的父母，他们给我买的这台写作用的笔记本将近三年多来一直没出大故障，伴随我的学习，生活。

B 已知问题

- [Hobc] 的 Figure 34 的程序和源文件中差一个 $-$ 号。
- 本文给出的预览程序不能够正确显示 图 31 内容。

参考文献

- [HE] HE Li. METAPOST使用指南. Available from: <http://bbs.ctex.org/forums/index.php?act=Attach&type=post&id=17848>.
- [Hec] André Heck. Learning METAPOST By Doing. Available from: <http://remote.science.uva.nl/~heck/Courses/mptut.pdf>. 3.3.3, 4.1
- [Hoba] John D. Hobby. Drawing graphs with metapost. Available from: <http://www.tug.org/docs/metapost/mpgraph.pdf>.
- [Hobb] John D. Hobby. The metapost system. Available from: <http://www.tug.org/docs/metapost/mpintro.pdf>.
- [Hobc] John D. Hobby. A user's manual for metapost. Available from: <http://www.tug.org/docs/metapost/mpman.pdf>. 1, 2.4, 2.4, 2.7, 3.3.2, 3.3.3, 3.3.4, 3.3.5, 3.3.5, 4.3, 4.4, 4.5, B
- [Knu] Donald E. Knuth. *The Art Of Computer Programming*. Addison Wesley. 1
- [Knu91] Donald E. Knuth. *The T_EXBook*. Addison Wesley, 1991.
- [Zoo] Vincent Zoonekynd. Available from: <http://www.math.jussieu.fr/~zoonek/LaTeX/Metapost/metapost.html>. 4.5

索引

#@, 23
&, 14
*, 8
**, 8
+, 8
++, 8
+-+, 8
-, 8
--, 7
.., 11
... , 12
/, 8
:=, 8
<, 19
<=, 19
<>, 19
=, 19
>, 19
>=, 19
@, 23
@#, 23
%, 8
off, 29
on, 29

abs, 8
addto, 15
ahangle, 33
ahlength, 33
and, 19
angle, 30

beginfig, 7
begingroup, 20
beveled, 31
black, 10
blue, 10
bluepart, 11
bp, 8
btex, 15
butt, 21
Cantor集, 17

cc, 8
char, 14
cm, 8
color, 10
curl, 12
currentpen, 32
currentpicture, 15
cutafter, 33
cycle, 7

dashed, 28
dashpattern, 29
lstinlinedashpattern, 27
dd, 8
decimal, 14
def, 12
def, 20
dir, 12
direction of, 34
div, 8
dotlabel, 23
dotlabels, 23
dotprod, 9, 24
down, 12
downto, 19
draw, 7
drawarrow, 9
drawdblarrow, 32

else, 19
elseif, 19
end, 7
enddef, 12
endfig, 7
endfor, 9
endgroup, 20
epsilon, 8
etex, 15
evenly, 12
evenly, 28
exitif, 20
exitunless, 20

- expr, 21
- false, 19
- fi, 19
- fill, 12
- filldraw, 16
- for, 9
- forever, 19
- forsuffixes, 20, 23
- fullcircle, 29
- gobble, 33
- green, 10
- greenpart, 11
- grouping, 20
- hide, 19, 21
- if, 19
- in, 8
- infinity, 32
- infont, 15
- interim, 21
- intersectionpoint, 34
- Koch曲线, 18
- label, 15
 - label, 15
 - label.lft, 15
- left, 12
- length, 33
- linecap, 21
- llcorner, 29
- makepath, 14
- makepen, 14
- mexp, 8
- mitered, 31
- miterlimit, 32
- mlog, 8
- mm, 8
- mod, 8
- normaldeviate, 9
- not, 19
- nullpicture, 29
- numeric, 8
- oct, 14
- of, 24
- of operator, 24
- operator, 23
- or, 19
- origin, 23
- pair, 10
- pair, 9
- path, 11
- pc, 8
- pen, 14
- pen, 14
- pencircle, 9
- pickup, 9
- picture, 15
- point of, 33
- primary, 24
- primarydef, 24
- pt, 8
- readfrom, 15
- red, 10
- redpart, 11
- reverse, 32
- right, 12
- rotated, 27
- round, 11
- rounded, 21
- save, 20
- scaled, 9
- secondarydef, 24
- secondary, 24
- shifted, 10
- show, 21
- squared, 30
- step, 19
- str, 23
- string, 14
- string, 14

substring of, 14
suffix, 21
suffix, 18

tension, 12
tertiary, 24
tertiarydef, 24
text, 21
tracingonline, 10
transform, 16
transformed, 16
true, 19

uniformdeviate, 10
univector, 34
until, 19
up, 12
upto, 9
urcorner, 29

vardef, 22

warningcheck, 8
whatever, 10, 20
white, 10
withcolor, 10
withdots, 28
withpen, 14
write to, 15

xpart, 10, 16
xscaled, 27
xxpart, 16
xypart, 16

ypart, 10, 16
yscaled, 27
ypart, 16
yypart, 16

z, 23

后缀, 23

定比分点, 9
线型, 27

数组, 17

算符, 23